Koneoppimisen soveltaminen graafihakualgoritmien ohjaamisessa

Joel Luukka

Tampereen yliopisto
Luonnontieteiden tiedekunta
Tietojenkäsittelyoppi
Pro gradu -tutkielma
Ohjaaja: Jorma Laurikkala
Joulukuu 2017

Tampereen yliopisto
Luonnontieteiden tiedekunta
Tietojenkäsittelyoppi
Joel Luukka: Koneoppimisen soveltaminen graafihakualgoritmien ohjaamisessa
Pro gradu -tutkielma, 50 sivua
Joulukuu 2017

Graafirakenteita käytetään monissa eri sovelluksissa kuvaamaan niiden toiminta-ympäristöjen lainalaisuuksia. Graafien laajan sovelluskentän vuoksi graafihaku-algoritmit ovat keskeisessä roolissa sovellusten toteutuksessa. Graafihakuongelmaan on kehitetty erittäin hyvin toimivia algoritmeja, mutta tietoaineistojen kasvaminen ja sovelluskohteiden monimutkaistuminen asettavat yhä korkeampia tehokkuus-vaatimuksia graafihakualgoritmeille.

Graafihakumenetelmät ovat karkeasti jaettavissa kahteen joukkoon: heikkoihin hakumenetelmiin, jotka ovat suoritettavissa missä tahansa graafissa, ja heuristisiin hakumenetelmiin, jotka käyttävät sovellus-, hakutehtävä- ja graafikohtaista tietoa tehostaakseen hakua. Heikot hakumenetelmät ovat toimintavarmoja, mutta suorittavat paljon ylimääräisiä toimenpiteitä haun aikana, mikä tuottaa tehottomuutta. Heuristiset hakumenetelmät pystyvät välttämään osan ylimääräisistä vaiheista, mutta heuristisia hakuja on mahdollista räätälöidä vain rajattuun määrään sovelluskohteita.

Tässä tutkielmassa esittelen uuden koneoppimista käyttävän graafihaku-menetelmän, jonka oppimistehtävänä on tuottaa malli, joka ohjaa hakua heuristisen hakumenetelmien tapaan. Menetelmän oppiva osa perustuu vahvistusoppimiseen, jonka avulla voidaan purkaa tehokkaiden hakumenetelmien asettamia vaatimuksia graafien kuvaukseen ja laajentaa siten näiden hakumenetelmien sovelluskenttää.

Oppivan haun toimintaa mitattiin kahdessa kokeessa. Ensimmäisessä kokeessa oppivaa hakua verrattiin tehokkaaseen heuristiseen hakumenetelmään sille soveltuvissa hakutehtävissä. Toisessa kokeessa oppivan haun toimintaa tutkittiin heuristisille hauille sopimattomassa hakutehtävässä, jossa oppivan haun vertailukohtana käytettiin yksinkertaista leveyshakua.

Kokeiden tulosten perusteella oppivaa hakua voidaan pitää hyvänä keinona toteuttaa tehokkaampia hakumenetelmiä sovelluskohteissa, joissa heuristista hakua ei ole mahdollista käyttää. Tutkielmassa esitetty oppiva haku vaatii kuitenkin suuren muistimäärän sisäisen mallinsa ylläpitämiseen, mikä on otettava huomioon oppivan haun soveltamisessa ja jatkokehityksessä.

Avainsanat ja -sanonnat: graafit, graafihaku, koneoppiminen, vahvistusoppiminen

Sisällys

1. Johdanto

Graafirakenteet tarjoavat yleisen mallin jolla voidaan kuvata erilaisia suhteita ja prosesseja eri sovellusalueilla. Arkikielen sanat *verkko* ja *verkosto* tarkoittavat usein konkreettista tai abstraktia rakennetta, joka voidaan kuvata graafirakenteena. Esimerkiksi tie-, viemäri- ja tietoliikenneverkot ovat osa yhteiskunnan infrastruktuuria, jotka koostuvat geografisista pisteistä joiden välillä vallitsee jokin suhde. Abstraktimmalla tasolla esimerkiksi sosiaalinen verkosto voi kuvata ihmisiä ja ihmissuhteita, joista voidaan koostaa valtavia sosiaalisia suhteita kuvaavia graafeja. Graafirakenteet ovat usein kuitenkin niin monimutkaisia, että niistä haetaan erilaisia alirakenteita tiedon tuottamiseksi. Yksinkertaisimmillaan haettava alirakenne on yksittäinen solmu tai solmujen välinen yhteys, ja jos välitöntä yhteyttä solmujen välillä ei ole, voidaan graafista etsiä polkua annettujen solmujen välillä.

Graafirakenteiden tapaan myös graafihakualgoritmit (2. luku) ovat sovellettavissa hyvin erilaisiin ongelmiin. Graafihakualgoritmien sovelluskenttä on erittäin laaja ja niitä käytetään usein tarkoin määritettyihin tehtäviin. Niillä on myös paikkansa yleisemmän tason sovelluksissa, kuten logiikkaohjelmoinnin suoritusmallin keskeisenä toimintamallina [Tamir & Kandel, 1995], tietokonelaitteiston ja ohjelmiston perusarkkitehtuurissa [Treleaven *et al.*, 2005] ja tekoälymenetelmien päättelymallien toteutuksessa [Thornton & du Boulay, 1998]. Logiikkaohjelmoinnissa käytetyt symboliset ohjelmointikielet, kuten Prolog, perustavat suoritusmallinsa faktoista ja predikaateista muodostettuun tila-avaruuspuuhun. Ratkaisuja ohjelmalle esitettyihin kyselyihin etsitään tästä puusta yksinkertaisen syvyyshaun avulla [Tamir & Kandel, 1995]. Tekoälymenetelmien kehittämiseksi ja niiden sovellusten yleistymistä vauhdittamaan on tutkittu tietokonearkkitehtuureja, jotka tukevat paremmin symbolisia laskentamenetelmiä [Treleaven *et al.*, 2005]. Näin siis myös graafit ja graafihakumenetelmät voisivat olla keskeinen osa tietokonearkkitehtuureja samoin kuin pinorakenteet ovat nykytietokoneiden arkkitehtuurissa.

Thornton ja du Boulay [1998] esittävät, että kaikkien tekoälysovellusten voidaan nähdä suorittavan jonkinlaista graafihakua, jonka tarkoituksena on löytää polku lähtötilan ja tavoitetilan välillä. Esityksensä havainnollistamiseksi Thornton ja du Boulay antavat esimerkkejä eri tekoälymenetelmistä ja niiden toiminnasta, kuvaavat menetelmän toimintaympäristön graafina ja suorittavat yksinkertaisen hakumenetelmän graafissa. Lopputulemana on, että graafissa suoritettu haku tuottaa saman tai samankaltaisen tuloksen kuin alkuperäinen tekoälymenetelmä. Havainto on erityisen mielenkiintoinen graafihakumenetelmien tutkimuksen näkökulmasta, sillä graafihakumenetelmien parantaminen voi potentiaalisesti tehostaa valtavaa määrää tekoälysovelluksia.

Graafihakualgoritmien avulla voidaan siis etsiä ratkaisuja hyvin erilaisiin ja haastaviin ongelmiin. Monessa sovelluksessa käytetään heikkoja graafihakualgoritmeja niiden toimintavarmuuden vuoksi. Heikot hakualgoritmit (3. luku) tutkivat graafin järjestelmällisesti, mikä takaa sen, että haku löytää ratkaisun hakuongelmaan. Heikot hakualgoritmit epäonnistuvat vain, jos ratkaisua hakutehtävään ei ole olemassa. Heikot graafihakualgoritmit vaativat kuitenkin paljon aika- ja muistiresursseja toimiakseen, ja muistin rajallisuus on käytännön haaste, joka voi myös osoittautua heikkojen hakumenetelmien epäonnistumisen syyksi.

Heikkoja hakualgoritmeja tehokkaammat, heuristiset graafihakualgoritmit (4. luku) pystyvät tehostamaan hakua hyödyntämällä hakutehtävään ja graafiin liittyvää tietoa [Pearl, 1984]. Heuristiset hakumenetelmät vaativat kuitenkin tarkkaa sovelluskohtaista suunnittelua, eikä jokaiseen hakutehtävään ole mahdollista toteuttaa heuristista hakua. Monessa graafihakumenetelmien sovelluskohteessa ollaankin käytetty heikkoja hakumenetelmiä, koska heuristista hakua ei ole ollut mahdollista toteuttaa kyseiseen hakuongelmaan.

Graafihakumenetelmien tehostaminen ja heuristisiin hakumenetelmiin liittyvien rajoitteiden purkaminen on kannattava tutkimuskohde hakumenetelmien laajan sovelluskentän ja erityisesti hakumenetelmien yleisen tason sovellusten vuoksi. Koska hakumenetelmät ovat keskenään jotakuinkin samankaltaisia, on monessa sovelluksessa periaatteessa mahdollista saavuttaa parempi tehokkuustaso vaihtamalla sovelluksen käyttämä hakumenetelmä tehokkaampaan. Tässä tutkielmassa pohdin mahdollisuutta tehostaa graafihakualgoritmeja vahvistusoppimisen [Sutton & Barto, 1998] mukaisen koneoppimismenetelmän avulla (5. luku). Vahvistusoppimisen etu on, että se vaatii toimintaympäristöään kuvaavan graafin solmuista vain yksilöivän tunnisteen, minkä vuoksi on mahdollista toteuttaa tehokkaampia graafihakumenetelmiä pienemmillä vaatimuksilla sovelluskohteen graafia kohtaan kuin mitä heuristiset hakumenetelmät vaativat.

Tässä tutkielmassa esitän uuden oppivan graafihakualgoritmin, joka pyrkii vahvistusoppimisen avulla oppimaan toimintaympäristönsä lainalaisuudet ja tuottamaan sisäisen mallin, joka ohjaa hakua heuristisen hakumenetelmien tapaan (6. luku). Oppivan hakualgoritmin tehokkuutta mitataan kahdessa kokeessa (7. luku). Ensimmäisessä oppivaa hakua käytetään satunnaisesti generoiduissa graafeissa suoritettavissa hakutehtävissä ja sitä verrataan heuristiseen hakualgoritmiin, jonka tiedetään toimivan tehokkaasti annetuissa tehtävissä. Toisessa kokeessa käytetään sosiaalista verkkoa kuvaavaa graafia, joka on alunperin koottu todellisista sosiaalisen median palvelun käyttäjistä [Leskovec & Krevl, 2014; McAuley & Leskovec, 2012].

Ensimmäisessä kokeessa oppivan haun tehokkuutta verrataan heuristisen A*-haun [Pearl, 1984] tehokkuuteen. Mittaustulokset osoittavat, että oppiva hakualgoritmi on verrattain tehokas. Vaikka oppiva haku ei täysin saavuta A*-haun tehokkuustasoa annettujen aikarajoitteiden puitteissa, antavat mittaustulokset viitteitä siitä, että oppiva haku voi saavuttaa A*-haun tehokkuustason, kun haku suoritetaan suurissa graafeissa. Tämän lisäksi oppiva haku vaatii vähemmän sovelluskohtaista mukauttamista kuin heuristiset hakualgoritmit.

Toisessa kokeessa oppivan haun tehokkuutta mitataan sosiaalisia suhteita kuvaavassa graafissa. Koska tehokasta heuristista hakua on vaikeaa toteuttaa sosiaalisten verkkojen muodostamiin graafeihin, käytetään vertailukohtana leveyshakualgoritmia. Oppiva haku pystyy löytämään ratkaisut annettuihin hakutehtäviin ja osoittautuu erittäin tehokkaaksi hakumenetelmäksi verrattuna leveyshakuun.

Kokeiden tulokset ovat lupaavia, sillä ne osoittavat koneoppimisen tuovan heuristisiten hakujen tasoisen hyödyn sovelluskohteisiin, joissa heuristiset haut eivät ole helposti sovellettavissa. Koneoppimisen soveltaminen graafihakuun tuo kuitenkin algoritmien kehitykseen uudenlaisia ongelmia, jotka kumpuavat oppimisen toteutustavasta. Tutkielmassa esitetyn oppivan haun sisäisen mallin ylläpitäminen asettaa melko korkeat vaatimukset käytettävissä olevan muistin määrälle.

Ongelma on kuitenkin hyvin tunnettu, joten keinoja ongelman hillitsemiseksi voidaan hakea vahvistusoppimisen tutkimuksesta.

2. Graafihakualgoritmit

Graafihakualgoritmit ovat mielenkiintoisia graafien monipuolisten käyttötapojen vuoksi; graafien avulla voidaan mallintaa lähes mitä tahansa järjestelmää. Graafin rakennetta tutkittaessa mielenkiinnon kohteina ovat usein graafin eri osien väliset yhteydet ja polut, joita graafihakualgoritmit pyrkivät löytämään. Graafeja ja graafihakualgoritmeja on sovellettu muun muassa reitinetsintään tieverkostossa [Guzolek & Koch, 1989], kuvadatan käsittelyyn [Silvela & Portillo, 2001] ja analyysiin [Stamatelos *et al.*, 2014], metsäpalojen mallintamiseen [Stepanov & Smith, 2012], robottien ohjaukseen [Chen *et al.,* 2014], sosiaalisen verkoston analyysiin [Wilson *et al.*, 2009; Beamer *et al.*, 2013, Fay, 2016] ja internetkäyttäytymisen analyysiin [Nanopoulos & Manolopoulos, 2001; Wang & Lee, 2011]. Vaikka graafeja ja graafihakuja on sovellettu hyvin erilaisiin sovelluskohteisiin, pysyvät graafien perusmuoto ja hakualgoritmit pääosin samanlaisina. Tästä syystä graafihakualgoritmit ovat erinomainen työkalu erilaisissa tekoälysovelluksissa.

Graafit koostuvat *solmuista* V ja solmuja yhdistävistä *kaarista* E. Kaaret voivat olla *suunnattuja,* jolloin yhteys solmujen välillä on yksisuuntainen, tai *suuntaamattomia*, molempiin suuntiin kulkevia yhteyksiä. Suuntaamaton kaari voidaan myös nähdä kahtena suunnattuna kaarena, jotka yhdistävät samat solmut, mutta kulkevat vastakkaisiin suuntiin. Solmun *naapureiksi* kutsutaan sitä solmujen joukkoa, jonka alkiot ovat välittömässä yhteydessä tutkittavaan solmuun. Toisin sanoen solmun $v \in V$ naapurusto saadaan muodostamalla joukko $V' \subseteq V$, missä $(v, v') \in E$ ja $v' \in V'$.

Puurakenteet ovat graafien erikoistapauksia. Puurakenteessa on yksi *juurisolmu*, *sisäsolmuja* ja *lehtisolmuja*. Kaikilla solmuilla paitsi juurisolmulla on yksi *vanhempi* ja kaikilla paitsi lehtisolmuilla on yksi tai useampi *lapsisolmu*. Juurisolmu voi olla myös lehtisolmu silloin, kun graafissa on vain yksi solmu. Solmut, joilla on sekä vanhempi että lapsisolmuja ovat sisäsolmuja.

Graafihakualgoritmien toiminta on usein helppo käsittää puurakenteissa, sillä lehtisolmut rajaavat hakualgoritmin kulkua ja estävät haun päätymästä silmukkaan, josta hakualgoritmi ei välttämättä pääse pois. Puurakenteissa toimivat kuitenkin samat hakualgoritmit kuin graafeissa, vaikka puurakenteissa suoritetut hakualgoritmit eivät tarvitsekaan silmukoiden havaitsemista ja käsittelyä samalla tavalla kuin graafirakenteissa suoritetut hakualgoritmit.

Graafihakualgoritmit jaetaan usein *heikkoihin*- ja *heuristisiin* hakualgoritmeihin. Heikot hakualgoritmit perustuvat tietyn tietorakenteen, kuten pinon tai jonon toimintaan ja tutkivat järjestelmällisesti koko hakuavaruuden. Heuristiset hakualgoritmit hyödyntävät hakutehtävään liittyvää tietoa ja pyrkivät siten tuottamaan ratkaisun hakutehtävään heikkoja hakualgoritmeja tehokkaammin.

3. Heikot hakualgoritmit

Yksinkertaisimmat hakualgoritmit ylläpitävät listaa avoimista solmuista yksinkertaisessa tietorakenteessa. Solmua pidetään *avoimena* silloin, kun se on algoritmin edetessä havaittu, mutta sitä ei vielä olla käsitelty. Haun alkaessa avointen solmujen listalla on vain lähtösolmu. Listalta

valitaan yksi solmu, jota tutkitaan hakutehtävän lopetusehdon mukaan. Jos solmu ei täytä lopetusehtoa, se *laajennetaan,* eli haetaan kaikki solmun naapurustoon kuuluvat solmut. Naapuruston solmujen sanotaan *generoituneen,* ja ne lisätään avointen solmujen listaan. Laajentamisen jälkeen solmun käsittely päättyy, ja sitä kutsutaan *suljetuksi*. Tämän jälkeen valitaan uusi solmu avointen solmujen listalta laajennettavaksi. Tätä jatketaan kunnes avointen solmujen lista on tyhjä tai jokin muu lopetusehto täyttyy.

Puu- ja graafirakenteesta hakemiseen käytetyt heikot hakualgoritmit perustuvat usein syvyys- ja leveyshakuun [Kreher & Stinson, 1999]. Puurakennetta käsiteltäessä *syvyyshaku* tutkii puun järjestelmällisesti juuresta lehtisolmuun asti ennen kuin se palaa tutkimaan muita puun haaroja. Käytännössä algoritmi käsittelee avointen solmujen listaa pinona, eli viimeksi havaitut solmut käsitellään ensin. Syvyyshaku on kuvattu algoritmissa 1 [Pearl, 1984].

```
 1| AvoinLista ← aloitussolmu
 2| Ratkaisu ← tyhjä lista
 3| Toista kunnes AvoinLista on tyhjä
 4|    Poista ensimmäinen solmu s listasta AvoinLista
 5|    UudetSolmut ← laajenna(s)
 6|    Toista listan UudetSolmut solmuille s'
 7|       s'.vanhempi ← s
 8|    Lisää listan UudetSolmut solmut listan AvoinLista alkuun
 9|    Jos listan UudetSolmut s' on kohdesolmu
10|       Toista kunnes s' on määrittelemätön
11|          Lisää s' listan Ratkaisu alkuun
12|          s' ← s'.vanhempi
13|       Palauta(Ratkaisu)
```

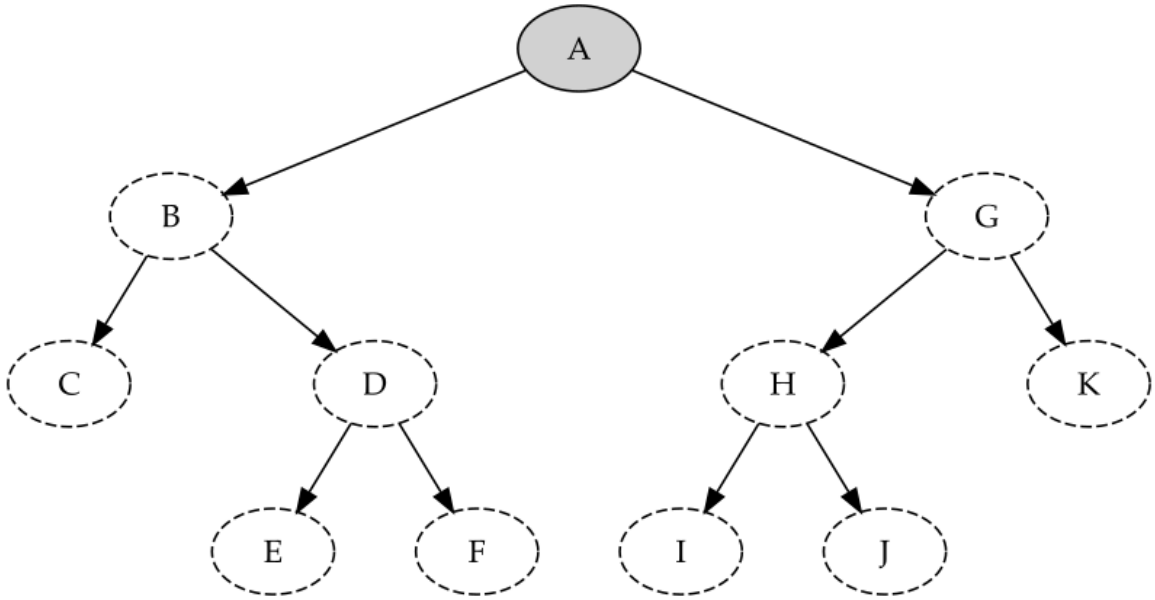
Algoritmi 1. Syvyyshaku

Pearl [1984] esittää syvyyshausta muistitehokkaamman version, jossa laajennettavan solmun naapurustosta generoidaan vain yksi solmu kerrallaan sen sijaan että koko naapurusto generoidaan kerralla. Tällöin laajennettava solmu *s* poistetaan avointen solmujen listalta vasta, kun kaikki sen lapsisolmut on generoitu. Solmua laajennettaessa solmuun *s* merkitään tieto generoidusta solmusta *s'*, jotta samaa lapsisolmua ei generoida uudelleen. Kun kaikki solmun *s* lapsisolmut on merkitty, voidaan solmu *s* poistaa avointen solmujen listalta, jolloin haku voi jatkua solmua *s* edeltävistä solmuista. Algoritmi käyttää muistia säästäväisemmin, koska avointen solmujen listaan lisätään solmuja vasta sitten, kun uutta solmua aletaan käsittelemään.

Puurakenteessa syvyyshaku alkaa juurisolmusta, jonka algoritmi tutkii lopetusehdon mukaan (kuva 1). Jos juurisolmu ei täytä lopetusehtoa, hakee algoritmi juurisolmun lapset, lisää ne avointen solmujen listan alkuun ja jatkaa solmujen käsittelyä avointen solmujen listan ensimmäisestä solmusta (kuva 2). Vasta lehtisolmun kohdalla avointen solmujen listaan ei lisätä uusia solmuja, jolloin haku palaa aiempaan puun haaraan ja valitsee sieltä toisen solmun käsiteltäväksi (kuva 3). Haku päättyy kun käsiteltäväksi valikoituu lopetusehdon mukainen solmu, jolloin voidaan

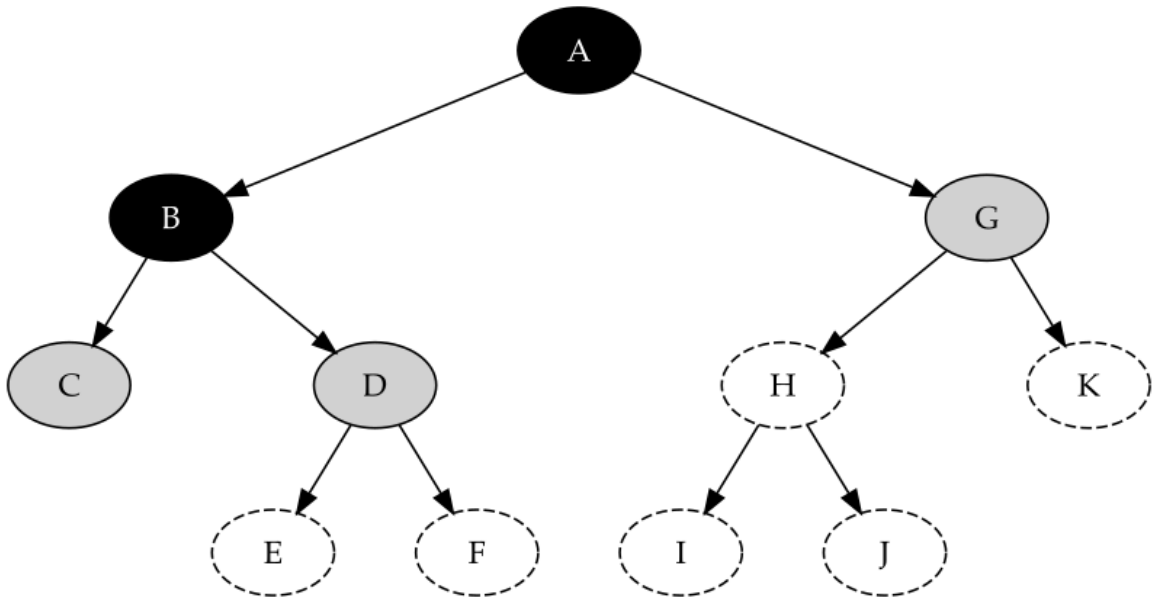
muodostaa polku hyväksytyn solmun ja juurisolmun välille hakemalla hyväksytyn solmun vanhemmat ja esivanhemmat aina juurisolmuun saakka (kuva 4). Haku voi myös päättyä avointen solmujen listan tyhjenemiseen, jolloin lopetusehdon mukaista solmua ei löydetty ja haku epäonnistuu (kuva 5). Syvyyshaku toimii hyvin matalissa hakupuissa, mutta hakupuun kerrosten määrän kasvaessa väärien puun haarojen tutkimiseen voi kulua paljon aikaa (kuva 6).

Avoin: [**A**]



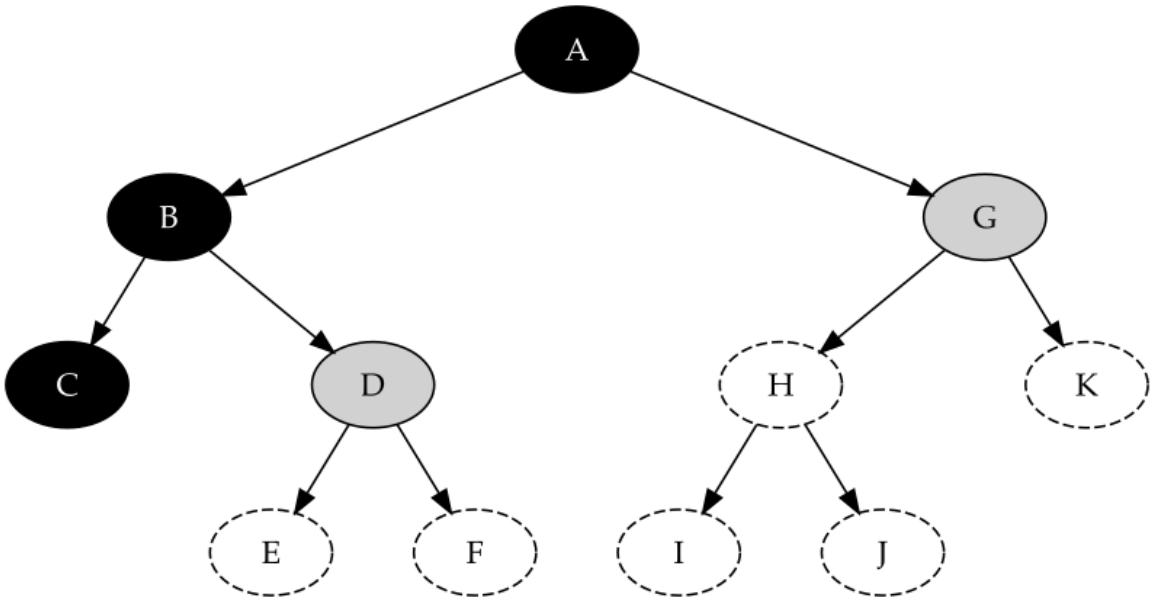
Kuva 1. Alkutilanne puurakenteessa suoritettavaan syvyyshakuun. Avoimessa listassa on vain juurisolmu ja kaikki muut solmut ovat vielä löytämättä.

Avoin: [**C**,**D**,G]



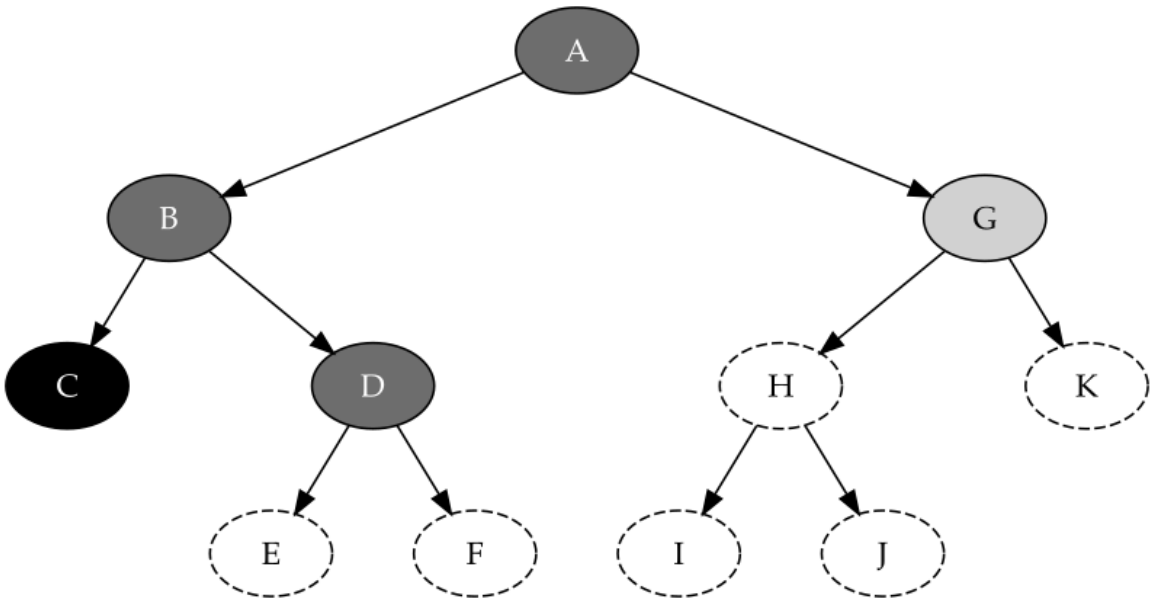
Kuva 2. Syvyyshaku lisää uudet solmut avointen solmujen listan alkuun, jolloin haku kulkee valitun puun haaran loppuun ennen kuin palaa aiempiin avattuihin solmuihin. Kuvassa on käsitelty solmu B, jonka lapset C ja D on lisätty avointen solmujen listan alkuun.

Avoin: [D,G]



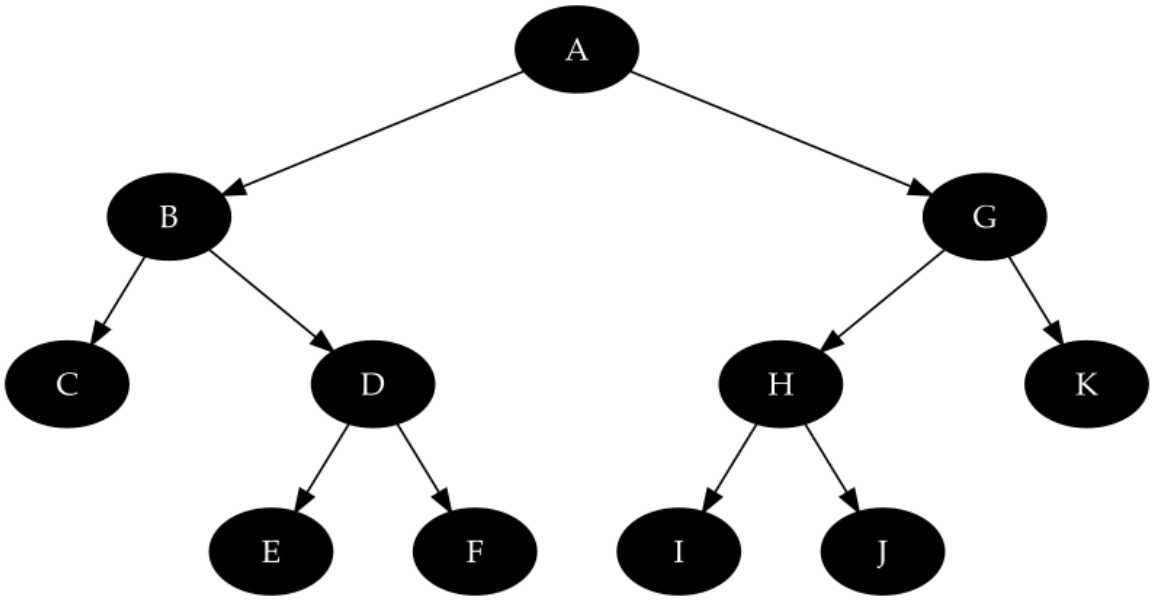
Kuva 3. Uusia solmuja ei lisätä avointen solmujen listaan, kun haku laajentaa lehtisolmun (C). Tällöin algoritmi palaa viimeksi avattuun solmuun (D).

Ratkaisu: [A,B,D]
Avoin: [G]

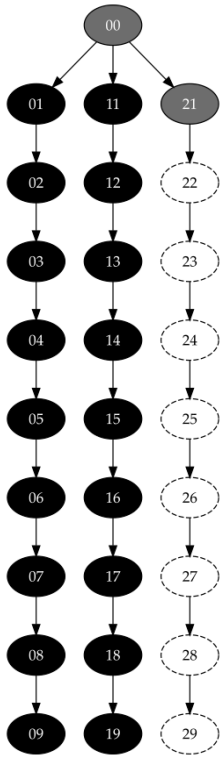


Kuva 4. Ratkaisupolku [A,B,D] voidaan muodostaa, kun lopetusehdon mukainen solmu (D) löydetään. Ratkaisupolku muodostetaan lopetusehdon mukaisen solmun vanhemmista.

Avoin: []



Kuva 5. Jos lopetusehdon mukaista solmua ei löydy, päätetään syvyyshaku kun avointen solmujen lista on tyhjä. Käytännössä syvyyshaku on tällöin tutkinut kaikki puun solmut.



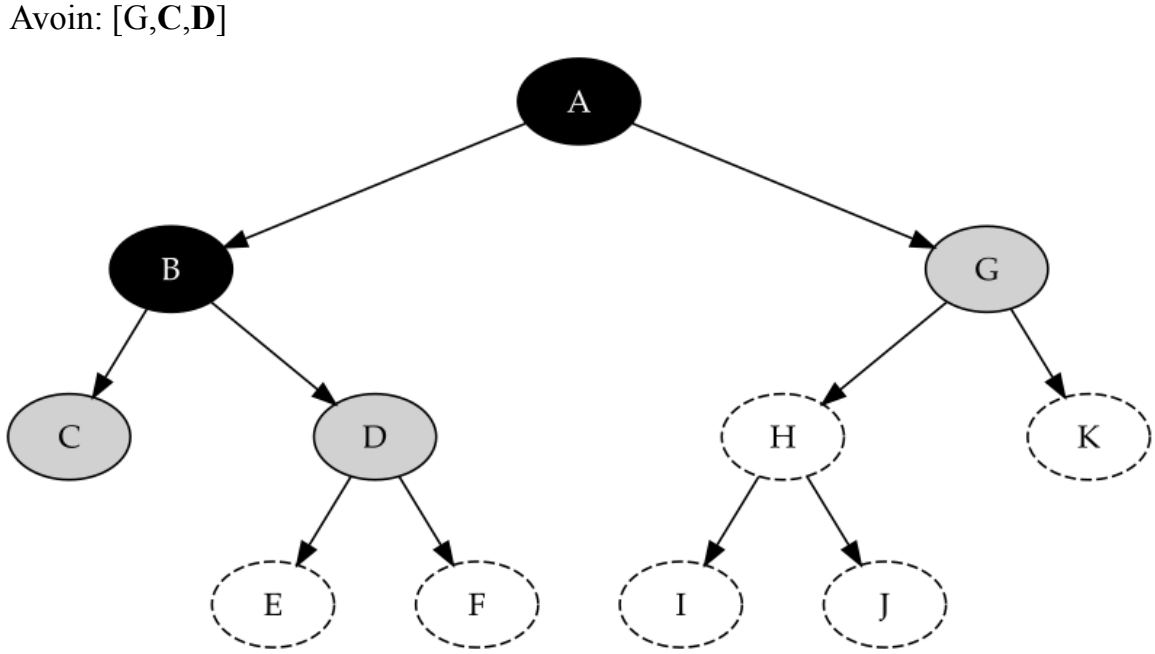
Kuva 6. Syvissä puurakenteissa syvyyshaku voi tutkia monta "väärää" puunhaaraa ennen kuin ratkaisusolmu laajennetaan. Tästä syystä syvyyshaku sopii hyvin mataliin puurakenteisiin.

Graafirakenteessa haku voi alkaa mistä tahansa graafin solmusta. Haun edetessä avoimien solmujen listaan lisätään kaikki käsitellyn solmun naapurisolmut. Koska graafissa ei ole yhtä selkeää haun lopettavien solmujen joukkoa, kuten lehtisolmut ovat puurakenteessa, saattaa hakualgoritmi jäädä laajentamaan solmuja, jotka yhdessä muodostavat silmukan. Tällainen ikuinen silmukka voidaan välttää pitämällä yllä suljettujen solmujen listaa, johon käsiteltävä solmu lisätään

laajentamisen jälkeen. Kun uusia solmuja generoidaan myöhemmin, voidaan tutkia kuuluuko generoitu solmu suljettujen solmujen joukkoon. Koska suljettujen solmujen listalla olevat solmut on käsitelty jo aiemmin, voidaan näiden solmujen uudelleenkäsittely ohittaa jättämällä ne lisäämättä avointen solmujen listalle.

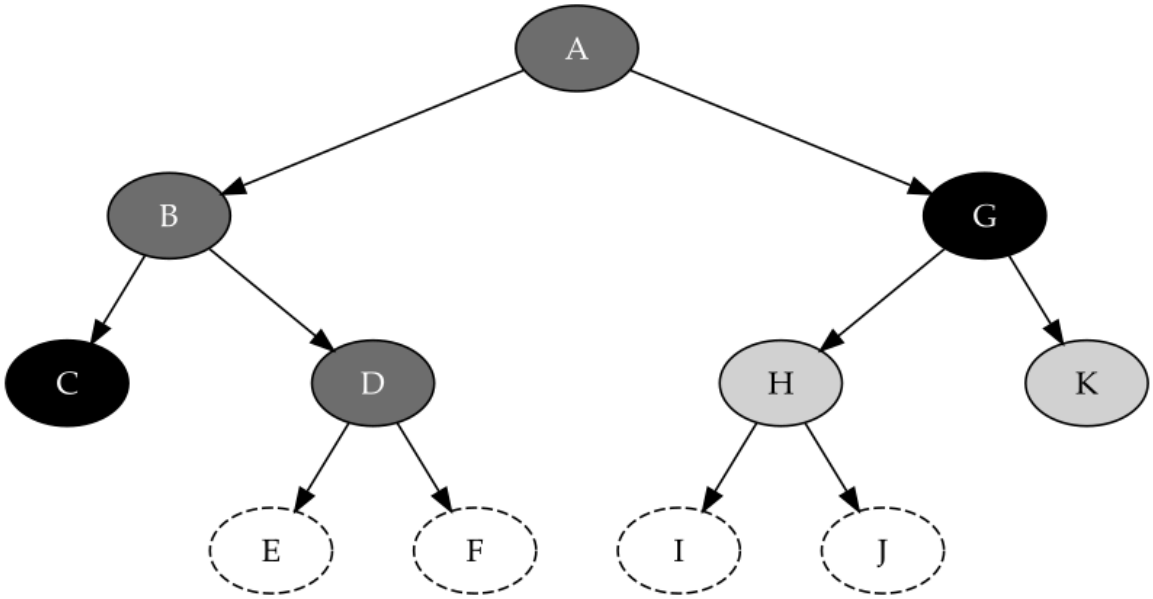
Leveyshaku on toinen heikko hakualgoritmi, joka tutkii solmut kerroksittain siten, että lähtösolmua lähinnä olevat solmut tutkitaan ensin. Algoritmi toimii pääosin samalla tavalla kuin syvyyshaku, mutta toisin kuin syvyyshaussa, avointen solmujen listaa käsitellään jonona, jolloin lapsisolmuja lisätään avoimien solmujen listan loppuun algoritmin 1 rivillä 8. Tästä syystä leveyshaku löytää ensin ratkaisut, joiden polku lähtösolmusta kohdesolmuun on lyhyin. Syvyyshaussa ratkaisujen polkujen pituutta tai järjestystä ei voida yhtä hyvin ennakoida. Leveyshaku vaatii kuitenkin enemmän muistia kuin syvyyshaku, sillä leveyshaussa ylläpidetään kaikkia mahdollisia hakupolkuja samaan aikaan. Syvyyshaussa puolestaan ylläpidetään vain yhtä hakupolkua kerrallaan.

Samoin kuin syvyyshaussa, leveyshaun alkaessa avointen solmujen lista sisältää vain lähtösolmun. Solmuja laajennettaessa uudet solmut lisätään kuitenkin avointen solmujen listan loppuun, jolloin haku alkaa toimia syvyyshausta poikkeavalla tavalla jo toisen solmun laajennuksen jälkeen (kuva 7). Leveyshaku laajentaa solmut kerros kerrokselta, minkä vuoksi kaikki ratkaisusolmun kerrosta alemmilla kerroksilla olevat solmut tulevat laajennetuksi ratkaisusolmun laajentamiseen mennessä (kuva 8). Leveyshaku onkin sopiva hakumenetelmä silloin kun hakupuu sisältää vähän haaroja, mutta runsashaaraisessa hakupuussa ratkaisun löytymiseen voi kulua monta laajennusvaihetta (kuva 9).

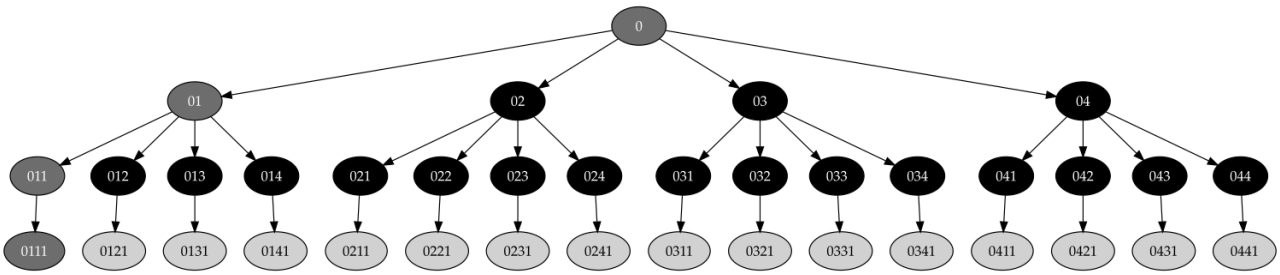


Kuva 7. Solmuja laajennettaessa leveyshaussa uudet solmut (C ja D) lisätään avoimien solmujen listan loppuun.

Ratkaisu: [A,B,D]
Avoin: [H,K]



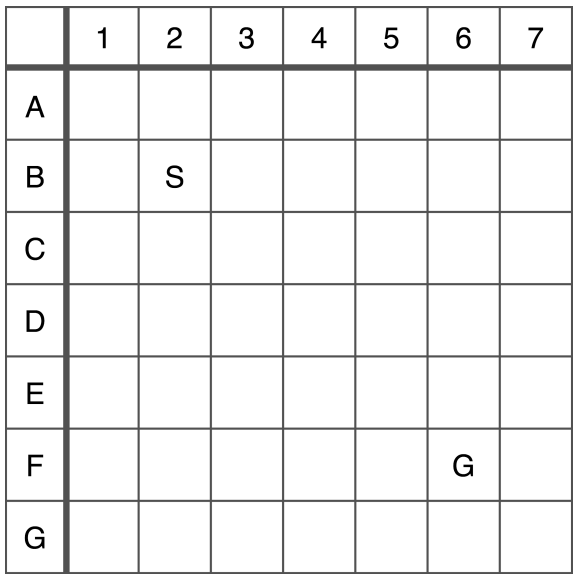
Kuva 8. Tilanne leveyshaun löytäessä ratkaisun hakutehtävään – kaikki ratkaisusolmua edeltävien kerrosten solmut on laajennettu, mutta seuraavien kerrosten solmuja ei ole käsitelty.



Kuva 9. Runsashaaraisessa puurakenteessa leveyshaku joutuu laajentamaan suuren määrän solmuja ennen ratkaisusolmun löytymistä.

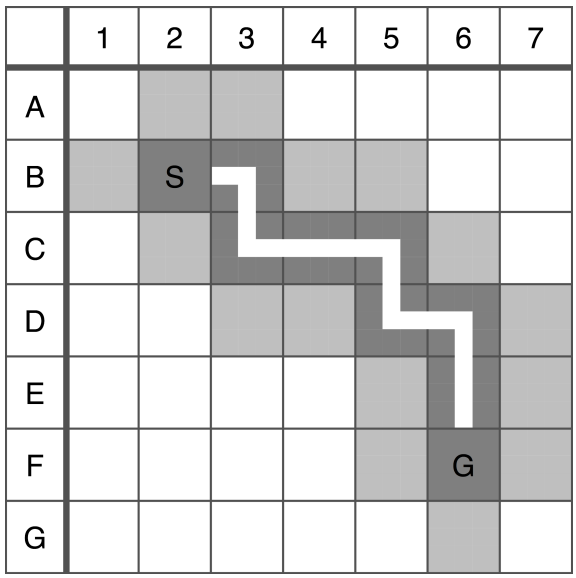
Algoritmien tehokkuuden mittarina käytetään algoritmin karsimien solmujen määrää [Pearl, 1984], sillä mitä vähemmän algoritmi laajentaa solmuja, sitä tehokkaammin algoritmi löytää ratkaisun. Tämän lisäksi löydetyn ratkaisun laatu otetaan huomioon algoritmeja verrattaessa.

Hakualgoritmien toiminnan havainnollistamiseksi graafeissa määritellään esimerkinomainen hakutehtävä 7×7-kokoisessa ruudukossa, jossa jokainen ruutu kuvaa graafin solmuja ja solmuista kulkee suunnattu kaari solmuihin, jotka ovat suoraan lähtösolmun ylä- tai alapuolella, oikealla tai vasemmalla. Näin siis solmujen väliset yhteydet ovat kaksisuuntaisia. Hakutehtävänä on löytää polku ruudusta B2 ruutuun F6. Kuvassa 10 on esitetty hakutehtävän lähtöasetelma.

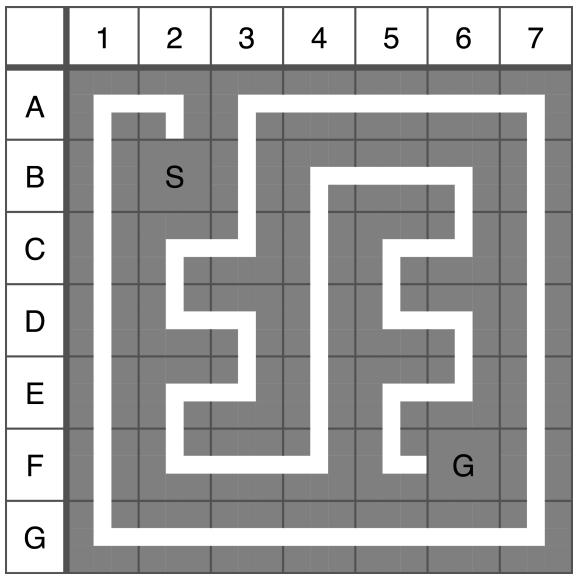


Kuva 10. Hakutehtävä lähtöasetelma. Ruutu *S* on hakutehtävän lähtösolmu ja ruutu *G* on hakutehtävän kohdesolmu.

Koska määritetty hakuongelma on yksinkertainen ja sen lainalaisuudet tunnettu, voidaan hakuongelmaan päätellä paras ja huonoin mahdollinen ratkaisu, joihin algoritmien tuottamia ratkaisuja voidaan verrata. Paras ratkaisu muodostuu hakupolusta, jossa siirrytään neljästi oikealle ja neljästi alas. Parhaita mahdollisia ratkaisuja on useita, koska siirtymien järjestyksellä ei ole merkitystä. Huonoin ratkaisu muodostuu hakupolusta, joka on käynyt jokaisessa graafin solmussa. Kuvassa 11 esitetään yksi parhaista mahdollisista ratkaisuista ja kuvassa 12 on esimerkki huonosta ratkaisusta.

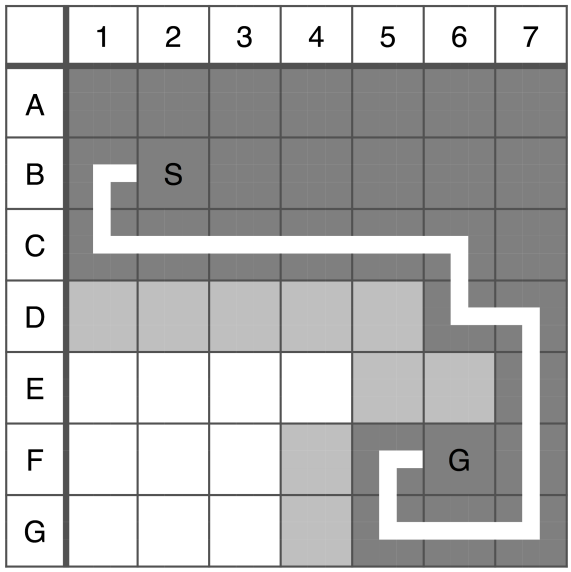


Kuva 11. Yksi parhaista mahdollisista ratkaisuista hakutehtävään. Löydetty polku on esitetty valkoisena polkuna. Tummat ruudut esittävät laajennettuja solmuja ja vaalean harmaat ruudut generoituja, avointen solmujen listaan lisättyjä solmuja.



Kuva 12. Yksi huonoimmista mahdollisista ratkaisuista hakutehtävään. Löydetty polku kulkee kaikkien graafin solmujen kautta.

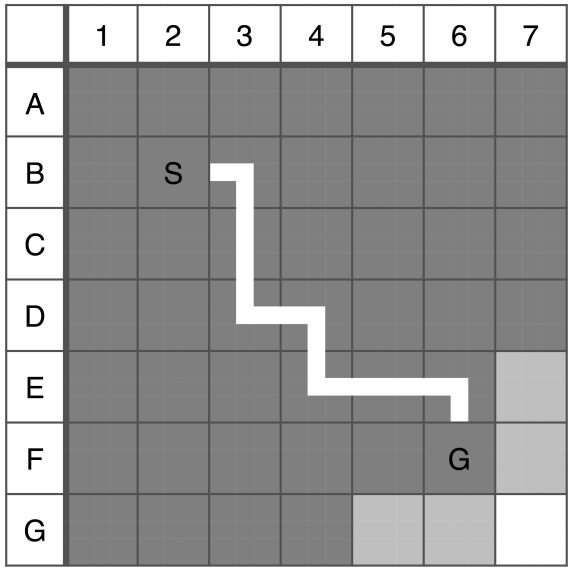
Syvyyshaku valitsee laajennettavaa solmua seuraavan solmun perustuen ensimmäiseen kaareen, jonka se löytää kulkiessaan laajennettavasta solmusta toiseen solmuun. Syvyyshaun etenemistä on siis vaikea ennustaa ilman ennakkotietoa kaarien käsittelyjärjestyksestä. Jotta hakutehtävässä ilmenisi syvyyshaun arvaamaton luonne, ovat graafirakenteen kaaret siinä satunnaisessa järjestyksessä. Kuvassa 13 on esitetty syvyyshaulla ensimmäinen löydetty ratkaisu.



Kuva 13. Syvyyshaku ensimmäisen ratkaisun löydyttyä.

Kuvasta 13 ilmenee, että vaikka syvyyshaku on onnistunut löytämään hyväksyttävän polun lähtösolmusta kohdesolmuun, ei polku kuitenkaan ole lyhyin mahdollinen ja hakualgoritmi on tutkinut ylimääräisiä solmuja ennen ratkaisun löytämistä. Tässä tapauksessa tarpeettomasti tutkitut solmut ilmenevät tummina ruutuina hakupolun yläpuolella.

Leveyshaku ei ole yhtä herkkä graafin sisäisen tietorakenteen esitysmuodolle kuin syvyyshaku. On kuitenkin odotettavissa, että leveyshaku on laajentanut pääosan hakutehtävän solmuista ensimmäisen ratkaisun löydyttyä. Tämä tilanne on esitetty kuvassa 14.



Kuva 14. Leveyshaku ensimmäisen ratkaisun löydyttyä.

Kuva 14 esittää leveyshaun löytämän ratkaisun hakuongelmaan. Löydetty polku on yksi lyhyimmistä mahdollisista poluista annettuun tehtävään, mutta algoritmi on tutkinut paljon ylimääräisiä solmuja ennen ratkaisun löytämistä.

Heikoilla hakualgoritmeilla on nimestään huolimatta laaja kirjo sovellusalueita, joissa ne ovat lähes korvaamattomia. Stamatelos [2014] tutkimusryhmineen sovelsi sekä syvyys- että leveyshakua luodessaan mallin syöpäkudoksen hiussuonistosta. Tässä sovelluksessa leveyshakua käytettiin muodostamaan hiussuonistosta suunnattu graafi, kun lähtötietona käytettiin suuntaamatonta graafia. Leveyshaun avulla suunnattu graafi saatiin mallintamaan oletettuja veren virtaussuuntia sillä oletuksella, että veri virtaa lyhimpiä mahdollisia polkuja pitkin. Suunnatussa graafissa suoritettua syvyyshakua käytettiin jaottelemaan graafi pienempiin segmentteihin.

Silvela ja Portillo [2001] käyttivät leveyshakua erilaisten kuvankäsittelyalgoritmien, kuten alueen täyttö-, eroosio- ja etäisyysmuunnosalgoritmien toteuttamiseen. Leveyshakua hyödyntävät algoritmit pystyivät kuvaa käsitellessään erottelemaan alueen, johon määrätty operaatio vaikuttaa, minkä vuoksi algoritmien tehokkuus oli moninkertaisesti parempi verrattuna algoritmeihin, jotka käsittelivät koko kuva-alueen.

Wilson tutkimusryhmineen [2009] käytti leveyshakua kootessaan Facebook-käyttäjistä aineiston, joka koostui käyttäjistä ja näiden välisestä vuorovaikutuksesta. Leveyshakua käytettiin tutkimusta varten toteutetun "ryömijän" (*crawler*) ohjauksessa, jonka tehtävänä oli kulkea käyttäjäprofiilien läpi keräten kustakin profiilista kommentti- ja kuvajakotietoja. Ryömijä kulki siis lähtöprofiilista kaikkiin profiiliin välittömästi yhteydessä oleviin profiileihin ja jatkoi näistä profiileista välittömästi niihin yhteydessä oleviin profiileihin leveyshaun mukaisesti.

Sovelluskohtaisten toteutusten lisäksi heikkoja hakualgoritmeja käytetään myös yleisemmissä konteksteissa. Syvyyshaku on keskeinen osa Prolog-logiikkaohjelmointikielen toteutusten suoritusmallia [Tamir & Kandel, 1995]. Ohjelmaan sisällytetyt faktat ja predikaatit muodostavat hakuavaruuden ja graafin, jossa käytännössä jokainen fakta ja predikaatti muodostaa graafin solmun ja jokaisesta graafin solmusta voidaan kulkea mihin tahansa solmuun. Prolog-ohjelmalle esitetty kysely määrittää syvyyshaulle lähtösolmun ja syvyyshakua jatketaan kunnes hakupolun

faktoista ja predikaateista muodostettu lause on tosi tai koko hakuavaruus on tutkittu ilman yhtäkään todeksi havaittua lausetta. Prologin suoritusmallin keskeiseksi hakualgoritmiksi on valittu syvyyshaku muistitehokkuuden vuoksi. Ohjelmoijalla on kuitenkin monia keinoja ja samalla vastuu hakupuun karsimisesta siten, että syvyyshaku toimii tehokkaasti ohjelman suorituksessa.

Leveyshakua on pyritty tehostamaan joko muokkaamalla itse algoritmia [Beamer *et al.*, 2013] tai haun kohteena olevaa graafia [Fay, 2016]. Beamer tutkimusryhmineen pyrki parantamaan leveyshaun tehokkuutta ylläpitämällä kahta hakurintamaa: ensimmäisen juurisolmuna toimii haun lähtösolmu ja toisen juurisolmuna haun kohdesolmu. Kahden rintaman avulla haun avointen solmujen lista ei kasva yhtä nopeasti kuin yhden rintaman hakualgoritmissa. Kahden rintaman leveyshaku näkee kuitenkin kaikki avoimet solmut yhtä sopivina kandidaatteina, kuten yhden rintaman leveyshakukin, minkä vuoksi riittävän pitkissä ja runsashaaraisissa hakupoluissa tulee vastaan samat ongelmat kuin yhden rintaman leveyshaussa.

Fayn [2016] lähestymistapa sosiaalisiin verkkoihin suoritetun leveyshaun tehostamiseen on jakaa graafi pienempiin osiin, joissa leveyshaku voidaan suorittaa rinnakkaisajona kussakin graafin osassa. Rinnakkaisajon ansiosta leveyshaku voidaan suorittaa erittäin nopeasti suuresta avointen solmujen määrästä huolimatta. Algoritmin kääntöpuolena on kuitenkin graafin osittamisprosessin resurssivaatimukset.

4. Heuristiset hakualgoritmit

Eräs keino pienentää heikkojen hakualgoritmien laskentakuormaa on käyttää hakuongelmaan liittyvää informaatiota avointen solmujen listan järjestämiseen siten, että haku ohjautuu kohti kohdesolmua. Heuristiset hakualgoritmit pyrkivät arvioimaan hakupolkua soveltuvuusfunktion avulla ja valitsemaan siten avoimista solmuista parhaan mahdollisen solmun, josta hakua jatketaan. Heuristiset hakualgoritmit siis järjestävät avoimien solmujen listan aina, kun siihen lisätään solmuja. Tämä johtaa siihen, että algoritmit eivät toimi sokeasti haun kohteena olevan graafin muodon johdattamana, kuten heikot hakualgoritmit. Soveltuvuusfunktio määräytyy sovelluskohteen ja valitun hakualgoritmin mukaan.

Kun hakutehtävään valitaan sopiva heuristinen hakualgoritmi, löydetään ratkaisu mahdollisesti tehokkaammin kuin heikoilla hakualgoritmeilla, sillä heuristiset hakualgoritmit ohjaavat algoritmia laajentamaan hakua annettuun hakutehtävään nähden lupaavimmalla solmulla. Hakujen käyttämät heuristiikat eivät kuitenkaan aina takaa tehokkaampaa hakua, mutta löytävät usein kelvollisia ratkaisuja.

Ahne paras ensin -hakualgoritmi on yksinkertaisin heuristinen haku, joka etsii polun kahden solmun välillä graafissa [Pearl, 1984]. Algoritmi valitsee laajennettavaksi sen solmun, joka ominaisuuksiensa perusteella havaitaan lupaavimmaksi laajennuskohteeksi. Ahneen algoritmin heikkoutena on riski juuttua graafin solmujen muodostamaan silmukkaan, josta algoritmilla ei ole keinoja päästä pois. A*-algoritmissa [Hart, Nilsson & Raphael, 1968] soveltuvuusfunktio koostetaan laskemalla yhteen jo kuljetun polun kustannusarvo ja arvio tulevan polun kustannusarvosta. Lisäämällä kuljetun polun kustannusarvo soveltuvuusfunktion arvoon saadaan haku kokeilemaan uusia polkuja, jos kuljetun polun kustannusarvo kasvaa liian suureksi. Tällöin

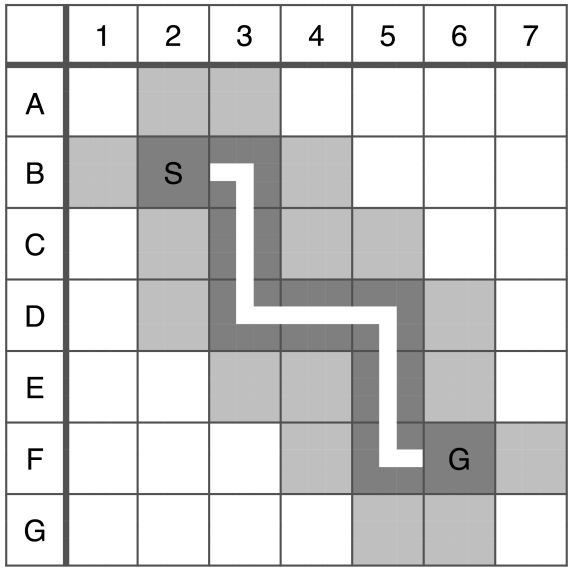
esimerkiksi solmujen muodostama silmukka kasvattaa kuljetun polun arvoa jokaisella laajennuksella, jolloin haku päätyy tutkimaan muitakin solmuja laajennettuaan silmukan solmuja tarpeeksi.

Ahne paras ensin -hakualgoritmi [Pearl, 1984] pyrkii ohjaamaan leveyshakua valitsemalla laajennettavaksi solmuja, jotka soveltuvuusfunktion mukaan ovat lupaavimpia. Menetelmä valitsee nimensä mukaisesti solmun tämän omien ominaisuuksien perusteella, eikä huomioi muita saatavilla olevia tai pääteltäviä tietoja, kuten solmuun johtaneen polun pituutta. Ahne paras ensin -haku on esitetty algoritmissa 2 [Pearl, 1984].

```
 1| AvoinLista ← aloitussolmu
 2| Ratkaisu ← tyhjä lista
 3| Toista kunnes AvoinLista on tyhjä
 4|    Poista listasta AvoinLista solmu s, jonka arvo f on pienin
 5|    UudetSolmut ← laajenna(s)
 6|    Toista listan UudetSolmut solmuille s'
 7|       s'.vanhempi ← s
 8|       laske arvo f(s') ja liitä arvo solmuun s'
 9|    Lisää listan UudetSolmut solmut listaan AvoinLista
10|    Jos solmu s on kohdesolmu
11|       Toista kunnes s on määrittelemätön
12|          Lisää s listan Ratkaisu alkuun
13|          s ← s.vanhempi
14|       Palauta(Ratkaisu)
```

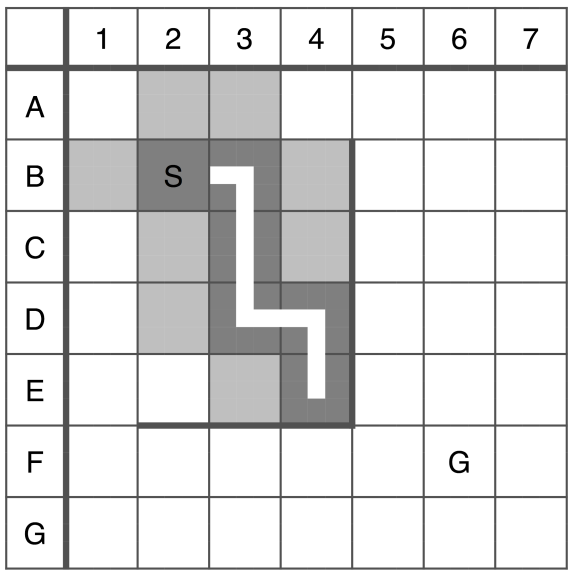
Algoritmi 2. Ahne paras ensin -haku

Aiemmin esitettyyn hakutehtävään suoritettu ahne paras ensin -haku tuottaa helposti parhaan mahdollisen ratkaisun, kun heuristiikkana käytetään solmujen välistä euklidista etäisyyttä. Tällöin soveltuvuusfunktio on muotoa $f(s) = Euclidean(s,G)$, missä $s \in S$ ja G on kohdesolmu. Kuvassa 15 on esitetty ahne paras ensin -haun löytämä ratkaisu hakutehtävään.



Kuva 15. Ahne paras ensin -haun löytämä ratkaisu hakutehtävään.

Algoritmin suorituksen yhteydessä on mahdollista, että haku jää tutkimaan samaa joukkoa solmuja, jos solmut muodostavat umpikujan tai silmukan, mutta ovat soveltuvuusfunktion mukaan lupaavampia kuin tavoitesolmuun vievän kiertopolun solmut. Algoritmi valitsee siis ahneesti soveltuvuusfunktion mukaan parhaan mahdollisen solmuvaihtoehdon kaikista avoimista solmuista riippumatta siitä onko solmut tutkittu jo aiemmin. Riski haun päätymiseen ikuiseen silmukkaan on algoritmin erityispiirre, joka on otettava huomioon soveltuvuusfunktion suunnittelussa, hakuongelman määrittelyssä ja algoritmin etenemisen analysoinnissa. Ahne paras ensin -haun heikkous voidaan havainnollistaa muokkaamalla hakutehtävän graafin sisältämiä kaaria siten, että ratkaisuissa on pakko siirtyä ainakin kerran poispäin kohdesolmusta. Kuvassa 16 on esitetty ahne paras ensin -haku, joka on jäänyt umpikujaan muokatussa hakutehtävässä.



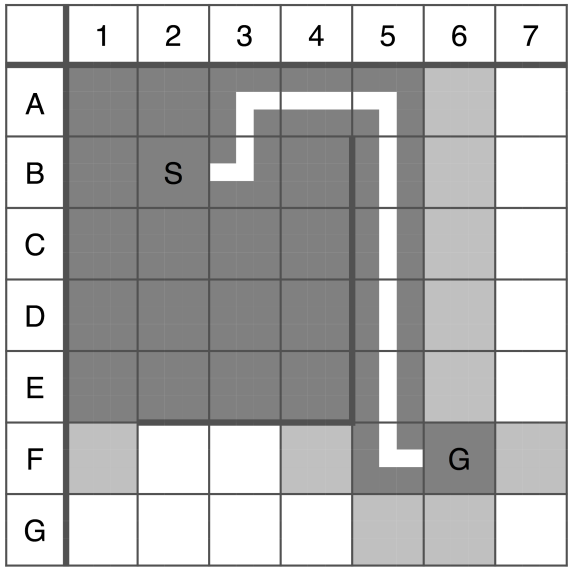
Kuva 16. Ahne paras ensin -haku muokatussa hakutehtävässä. Algoritmi jää kulkemaan ruutujen D4 ja E4 väliä, koska algoritmilla ei ole keinoa havaita kulkemiaan silmukoita. Muokatussa hakutehtävässä algoritmi ei voi ylittää paksua rajaa ruutujen välillä.

Algoritmin heuristiikka järjestää avointen solmujen listan aina siten, että kohdesolmusta poispäin kulkevia kaaria ei koskaan kuljeta. Haun päätyessä kuvan 16 osoittamaan umpikujaan ruudussa E4, voidaan hakua jatkaa vain ruutuihin E3 tai D4. Euklidisella etäisyydellä mitattuna ruutu D4 on ruutua E3 lähempänä kohdesolmua, joten haku kulkee askeleen takaisin hakupolkua pitkin. Laajentaessa jälleen ruutua D4 haku päätyy valitsemaan ruudun E4 kuten aiemmin. Ahne algoritmi ei siis pysty löytämään ratkaisua muokattuun hakutehtävään, koska sen heuristiikka ei kykene havaitsemaan ruutujen E4 ja D4 muodostamaa ansaa. Tästä huolimatta paras ensin -haku kiteyttää hyvin heuristisiten hakumenetelmien perusajatuksen; algoritmin käyttämä heuristiikka asettaa tutkittavat solmut paremmuusjärjestykseen, ja hakupuuta laajennetaan aina parhaasta mahdollisesta solmusta.

Yllä kuvatun kaltainen ikuinen silmukka on toki vältettävissä esimerkiksi ylläpitämällä suljettujen solmujen listaa, joka estää samojen solmujen laajentamisen yhä uudelleen. Suljettujen solmujen lista kuitenkin lisää algoritmin monimutkaisuutta ja muistivaatimusta, mikä on otettava huomioon algoritmin suunnittelussa ja käyttökohteen vaatimuksissa. Solmujen muodostamia umpikujia voidaan vältää myös muokkaamalla algoritmin käyttämää heuristiikkaa.

Ahne paras ensin -hakua voidaan parantaa lisäämällä kustannusarvo jokaiseen kaareen ja lisäämällä kuljetun polun yhteenlaskettu kustannus solmun etäisyysarvoon. Tällöin hakualgoritmi aloittaa kulkemalla samoin kuin haku ilman kustannusarvoa, mutta joutuessaan umpikujaan, kuljetun polun kustannus kasvaa ennen pitkää liian suureksi, jolloin haku siirtyy pois umpikujasta. Tämä yksinkertainen lisäys hakualgoritmin käyttämään heuristiikkaan takaa sen, että hakualgoritmi löytää aina kelvollisen ratkaisun.

A*-hakualgoritmi [Hart, Nilsson & Raphael, 1968] on yleinen muoto yllä kuvatusta kuljetun polun kustannusta laskevasta paras ensin -hausta. Algoritmin heuristiikka koostuu kahdesta osasta: kustannuksesta, joka syntyy kuljetusta polusta ja arviosta seuraavien solmujen sopivuudesta. Kuvassa 17 on esitetty A*-haun löytämä ratkaisu muokattuun hakutehtävään. Kuvasta nähdään, kuinka algoritmi tutkii koko umpikujan, mutta löytää lopulta reitin ulos umpikujasta ja ratkaisun hakutehtävään. Haun päätyessä ruutujen E4 ja D4 muodostamaan ansaan, hakupolun kokonaiskustannus nousee jokaisella edestakaisella laajennuksella. Kun umpikujaan päätyneen hakupolun kokonaiskustannus kasvaa riittävän suureksi, alkaa algoritmi tutkia muita vaihtoehtoisia solmuja. Kuvasta 17 ilmenee myös, että A*-haun löytämä ratkaisu on paras mahdollinen ratkaisu muokattuun hakutehtävään.



Kuva 17. A*-hakualgoritmin löytämä ratkaisu muokattuun hakutehtävään.

Heuristisen hakualgoritmien sovellusalueet ovat melko suppeat, sillä toimintavarman soveltuvuusfunktion laatiminen on vaikeaa. Reitinhaku fyysisessä ympäristössä on usein heuristisille hakumenetelmille sopiva ongelma, sillä soveltuvuusfunktion on tällöin arvotettava solmuja, jotka ovat lähempänä kohdesolmua tai nopeampien yhteyksien varrella. Esimerkiksi Guzolekin ja Kochin [1989] reitinsuunnittelujärjestelmässä on käytetty A*-hakualgoritmia optimaalisten reittien päättelyyn tieverkostossa. Monimutkaisemmissa sovelluksissa, kuten esimerkiksi visuaalisen sisällön käyttäjäkohtaisessa personoimisessa [Ferretti *et al.*, 2016] tai sosiaalisten verkkojen analyysissä [Fay, 2016], järkevän soveltuvuusfunktion laatiminen on kuitenkin erittäin hankalaa ellei mahdotonta.

5. Oppivat hakualgoritmit

A*-algoritmi osoittaa, että heuristiset haut pystyvät löytämään parhaan mahdollisen ratkaisun hakutehtävään, jos ratkaisu vain on löydettävissä. Tästä syystä A*-hakualgoritmi on sopiva moneen graafihakutehtävään. Heuristiset haut vaativat kuitenkin sovelluskohtaisen soveltuvuusfunktion, jota ei jokaiseen hakuongelmaan ole mahdollista toteuttaa. Heuristisen hakujen soveltaminen onkin tietyissä sovelluskohteissa epäkäytännöllistä, ellei mahdotonta toteuttaa, minkä vuoksi heikkoja hakumenetelmiä on sovellettu monimutkaisia ja suuria graafeja sisältävissä järjestelmissä. Tällaisissa järjestelmissä heikot hakualgoritmit ovat yksi keskeinen tehottomuuden aiheuttaja.

Keskeisin ero heikkojen ja heuristisen hakumenetelmien välillä on avointen solmujen listan järjestäminen, mikä ei lopulta ole kovin suuri muutos heikkojen hakujen algoritmiin. Heuristisen hakumenetelmien kehittäminen koneoppimisen avulla on ainakin ajatukseltaan luonnollinen ja yksinkertainen kehitysaskel; oppiva haku voisi yksinkertaisuudessaan olla heuristinen haku, jonka soveltuvuusfunktio annetaan koneoppimismenetelmän optimoitavaksi.

Oppivan graafihaun toteuttamiseen on käytetty useita eri menetelmiä, kuten solmujen piirteiden painottamista heuristisen soveltuvuusfunktion optimoimiseksi [Fink, 2007], heuristisen soveltuvuusfunktion asteittainen kehittäminen [Arfaee *et al.*, 2011] ja lineaarisen ohjelmoinnin

soveltaminen [Petrik & Zilberstein, 2008]. Fink [2007] toteutti oppivan hakumenetelmän, joka pyrkii oppimaan sopivan soveltuvuusfunktion, joka perustuu solmuihin liitettyihin ominaisuuksiin, kuten koordinaatteihin euklidisessa avaruudessa. Menetelmä tuottaa kahdesta solmusta piirrevektorin, jota painotetaan painoarvovektorin mukaan. Lopulta painotetusta piirrevektorista tuotetaan annetun funktion avulla etäisyysarvo kohdesolmuun. Painoarvovektorin arvojen oppimiseen käytetään lineaarista regressiomenetelmää. Toimiakseen menetelmä tarvitsee funktion, joka kuvaa solmuparin piirrevektoriksi, jonka arvot kuuluvat reaalilukujen joukkoon.

Arfaeen tutkimusryhmän [2011] oppivassa hakumenetelmässä heuristisen haun soveltuvuus-funktiota parannetaan asteittain. Oppimismenetelmälle annetaan alustava soveltuvuusfunktio ja suuri joukko hakutehtäviä, joita asteittaisen oppimisen aikana tulee ratkaista. Asteittainen oppiminen päätetään vasta, kun riittävän suuri osuus annetuista hakutehtävistä on ratkaistu. Varsinainen oppiminen on annettu neuroverkon tehtäväksi, jonka tehtävänä on kuvata sovelluskohteeseen räätälöity hahmotietokanta etäisyysarvoksi kohdesolmuun nähden.

Petrik ja Zilberstein [2008] käyttivät vahvistusoppimisessa usein käytettyä lineaarisen ohjelmoinnin menetelmää heuristisen soveltuvuusfunktion oppimiseen. Petrikin ja Zilbersteinin menetelmässä heuristisen soveltuvuusfunktion oppimista varten kerätään useita erilaisia hakutehtävien ratkaisupolkuja ja muodostetaan ratkaisupolkujen solmuista arvojoukot solmujen piirteistä. Lineaarista ohjelmointia käytetään piirrearvojoukon käsittelyyn, minkä lopputuloksena saadaan hakua ohjaava heuristinen soveltuvuusfunktio.

Vahvistusoppiminen on sopiva vaihtoehto oppivan hakumenetelmän toteutukseen, sillä sovellusympäristön kuvauksen oppiminen on osa vahvistusoppimisen toimintaa. Tuntematon tai muuttuva tieto ympäristöstä voidaan oppia hakujen suorituksen aikana, jolloin malli pysyy ajantasaisena dynaamisessakin ympäristössä. Vahvistusoppiminen vaatii myös melko vähän tietoa toimintaympäristöstään, joten sen perustoimintoja ei tarvitse räätälöidä jokaista sovelluskohdetta varten.

5.1. Vahvistusoppiminen

Vahvistusoppiminen on koneoppimismenetelmien joukko, jossa oppiminen perustuu agentin ja ympäristön väliseen vuorovaikutukseen [Sutton & Barto, 1998]. Agentit toimivat ympäristössään ja ympäristö antaa toiminnasta palautetta palkkion tai rangaistuksen muodossa. Agentin on opittava ympäristöltään saamasta palautteesta ja löydettävä toimintamalli, jota noudattamalla agentti maksimoi ympäristöltä saamansa *tuoton*, eli toiminnasta kertyvän palautteen summan. Koska agentti toimii stokastisesti ja pyrkii muodostamaan mahdollisimman tarkan mallin ympäristön antamasta palautteesta, voidaan agentin nähdä keräävän satunnaisotosta ympäristön palautefunktiosta. Joissain yhteyksissä vahvistusoppimiseen kuuluvia algoritmeja kutsutaankin otanta-algoritmeiksi [Andrieu, *et al*., 2003].

Klassinen esimerkki vahvistusoppimisen ongelmasta tunnetaan monikätisen rosvon ongelmana. Tässä Robbinsin [1952] määrittämässä ongelmassa on valittava yksi yhteinen otos useasta populaatiosta siten, että otoksen alkioiden arvojen yhteenlaskettu summa on mahdollisimman suuri tuntematta ennalta populaatioiden perusjakaumaa. Monikätisen rosvon metaforan mukaan

uhkapelaaja voi pelata useaa yksikätinen rosvo -rahapeliä. Ongelmana on päättää mitä peliä pelata ja kuinka monta kertaa. Lähtökohta ongelmassa on, että pelikoneiden todellista voittojakaumaa ei koskaan tiedetä, mutta koneiden voittojakauman oletetaan olevan erilainen kullekin pelikoneelle. Kun yhtäkään pelikonetta ei vielä ole pelattu, perustuu pelikoneiden pelaaminen puhtaaseen arvaukseen (taulukko 1). Havaitut voittojakaumat ovat epätarkkoja pienellä määrällä pelejä (taulukko 2), mutta mitä useammin pelaaja pelaa tietyllä pelikoneella, sitä tarkemman kuvan pelaaja saa pelikoneen antamien voittojen jakaumasta (taulukko 3). Pelikoneiden pelaaminen puhtaasti voittojakaumien selvittämiseksi voi kuitenkin käydä pelaajalle kalliiksi, joten pelaajan täytyy oppia hyödyntämään oppimiaan voittojakaumia maksimoidakseen voittonsa.

	Pelikone 1	**Pelikone 2**	**Pelikone 3**	**Pelikone 4**
Todellinen voittojakauma	0,9	0,7	0,3	0,5
Pelejä	0	0	0	0
Kokonaispalkkio	0	0	0	0
Havaittu voittojakauma (kokonaispalkkio / pelejä)	-	-	-	-

Taulukko 1. Monikätinen rosvo -ongelmassa pelikoneiden todelliset voittojakaumat ovat tuntemattomia, eikä niiden päättelyyn ole mitään keinoa ennen kuin pelejä on pelattu.

	Pelikone 1	**Pelikone 2**	**Pelikone 3**	**Pelikone 4**
Todellinen voittojakauma	0,9	0,7	0,3	0,5
Pelejä	8	4	6	2
Kokonaispalkkio	2	2	0	4
Havaittu voittojakauma (kokonaispalkkio / pelejä)	0,25	0,50	0,00	2,00

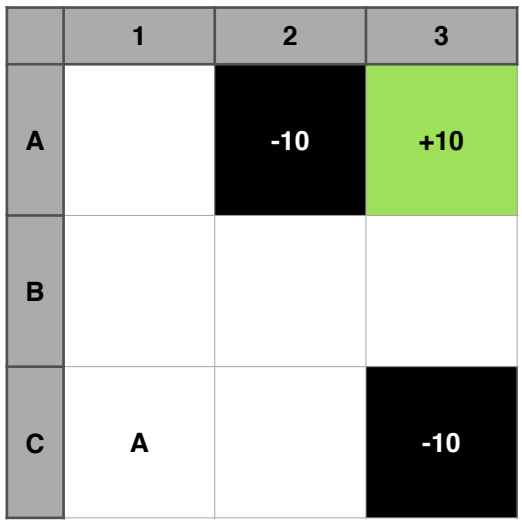
Taulukko 2. Kun jokaista pelikonetta on kokeiltu vähintään kerran, voidaan havaittujen voittojakaumien perusteella yrittää löytää parhaat voitot antava pelikone.

	Pelikone 1	Pelikone 2	Pelikone 3	Pelikone 4
Todellinen voittojakauma	0,9	0,7	0,3	0,5
Pelejä	20	22	12	16
Kokonaispalkkio	17	15	3	9
Havaittu voittojakauma (kokonaispalkkio / pelejä)	0,85	0,68	0,25	0,56

Taulukko 3. Kokeilumäärän kasvaessa havaitut voittojakaumat lähestyvät todellisia voittojakaumia, jolloin havaittujen jakaumien perusteella voidaan tehdä tarkempia arvioita tulevien pelien voitoista.

Monikätinen rosvo on yksinkertainen esimerkki vahvistusoppimisen toiminnasta. Vahvistusoppimisen nykysovellusten valossa monikätinen rosvo esittää kuitenkin vain pienen osan vahvistusoppimisen toiminnasta. Monikätisen rosvon kontekstissa pelaajan on aina valittava yksi pelikone samasta pelikoneiden joukosta, eivätkä pelaajan edelliset valinnat muuta pelaajalle esitettyä asetelmaa. Toisin sanoen ympäristöllä on vain yksi tila, jossa pelaaja toimii. Monissa vahvistusoppimisen sovelluksissa agentin valinnat kuitenkin muokkaavat ympäristön tilaa. Tällöin ei riitä, että pelaaja valitsee jokaista sille mahdollista toimintoa riittävän monta kertaa oppiakseen toimintaympäristönsä lainalaisuudet. Sen sijaan agentin on valittava jokainen sille mahdollinen toiminto riittävän monta kertaa kussakin ympäristön tilassa. Tällä tavoin agentin tulisi oppia toimintamalli, joka ohjaa agenttia valitsemaan parhaat toiminnot jokaisessa ympäristön tilassa ja joka ohjaa agentin parhaisiin ympäristön tiloihin.

Otetaan esimerkiksi 3×3-kokoinen ruudukko, jossa agentin on tarkoitus hakeutua yhdestä ruudukon kulmasta C1 vastakkaiseen kulmaan A3. Ruudukko toimii agentin ympäristönä ja jokainen ruutu vastaa yhtä ympäristön tilaa. Ruudukossa on kaksi "rotkoa", joihin kulkeminen johtaa negatiiviseen palautteeseen ja agentti palautetaan takaisin lähtöruutuun. Maaliruudun saavuttaminen puolestaan antaa agentille positiivisen palautteen ja päättää ohjelman suorituksen. Esimerkin ruudukon alkutilanne on annettu kuvassa 18.



Kuva 18. Vahvistusoppimisen esimerkkipelin alkutilanne.

Esimerkkipelissä agentti voi kunakin ajanhetkenä kulkea ylös, alas, vasemmalle tai oikealle. Kuvasta 18 voidaan päätellä, että toisin kuin monikätinen rosvo -pelissä, ruudukkopelissä yhden ainoan toiminnon valitseminen toistuvasti ei tuota optimaalista palkkiota, vaan paras toiminto tulee valita tilakohtaisesti. Vahvistusoppimiselle on ominaista, että oppimisjakson alussa agentti kulkee päämäärättömästi tilasta toiseen, kulkee kehää ja saa useasti negatiivisen palautteen rotkoruudusta. Oppimisen edetessä agentti oppii välttämään suoraan rotkoon johtavia toimintoja ja pystyy lopulta navigoimaan tarkasti rotkojen välistä maaliruutuun.

Vahvistusoppimiseen perustuva agentti perustaa toimintansa aiemmasta toiminnasta kartuttamaansa kokemukseen. Agentin toiminta ympäristössään on siis ikään kuin ulkoa opitun koreografian toistamista sen sijaan, että agentti osaisi havainnoida ympäristöään älykkäämmin kuin oppimisen alkuvaiheessa. Tämän vuoksi ympäristön ominaisuudet vaikuttavat osaltaan oppimistulokseen: yksinkertaisissa ja muuttumattomissa ympäristöissä agentin on helppo löytää optimi toimintamalli ympäristössä toimimiseen, mutta nopeasti tai satunnaisesti muuttuvissa ympäristöissä oppimistuloksen laatu voi jäädä heikoksi.

Vahvistusoppimisen toimintaympäristö kuvataan usein graafina, mikä mahdollistaa sen, että vahvistusoppiminen on sovellettavissa moniin optimointiongelmiin. Vahvistusoppimista on käytetty mukautuvan radiojärjestelmien toteuttamiseen [Lee *et al.*, 2016; Ling *et al.*, 2015], älykkään sähköverkon optimointiin [Mocanu *et al.*, 2016], monikamerajärjestelmien ohjaamiseen [Spurlock & Souvenir, 2016], robotiikkaan [Duguleana & Mogan, 2016; Fathinezhad *et al.*, 2016], virtuaalihahmojen käyttäytymisen ohjaamiseen [Feng & Tan, 2016], liikenteen ohjaukseen [Walraven *et al.*, 2016] ja internet selaimen näkymän personoimiseen [Ferretti *et al.*, 2016].

Agentin ympäristö koostuu tiloista S, joista yksi on kullakin ajanhetkellä vallitseva tila [Sutton & Barto, 1998]. Ympäristöistä voidaan muodostaa graafi, jossa ympäristön tilat kuvataan graafin solmuina ja tilojen väliset siirtymät solmujen välisinä kaarina. Ympäristön tiloihin on liitetty haluttavuusarvot $r \in \mathbb{R}$, jotka annetaan agentille palautteena agentin saapuessa kuhunkin tilaan. Ympäristöä voidaan kuvata Markov-prosessina, jossa tilojen väliset siirtymät nähdään luonteeltaan stokastisina. Agentti ei voi varmuudella tietää tulevaa tilaa, mutta ympäristöstä oppiminen on mahdollista, kun agentti yrittää laskea odotusarvoja suorittamiensa tilojen tuotolle.

Agentti pystyy havaitsemaan kullakin ajanhetkellä t vallitsevan ympäristön tilan $s_t \in S$ [Sutton & Barto, 1998]. Kussakin ympäristön tilassa s on valittavissa tietty joukko toimintoja $A(s)$. Agentin on valittava suoritettava toiminto $a \in A(s)$ ja suorittaessaan toiminnon nykyisessä ympäristön tilassa s_t, ympäristö vaihtaa vallitsevaa tilaa. Uuden tilan $s_{t+1} \in S$ mukainen palaute $r_{t+1} \in \mathbb{R}$ annetaan agentille ympäristön sisältämän palautefunktion $f(s) \rightarrow \mathbb{R}$ mukaan. Tämän jälkeen agentin on jälleen valittava uusi toiminto uudessa tilassa. Agentin ja ympäristön välinen vuorovaikutus jatkuu kunnes ympäristö siirtyy lopetustilaan tai muu lopetusehto täyttyy.

Agentin tehtävä on oppia toimintamalli π, jota noudattamalla saavutetaan paras mahdollinen tuotto ympäristössä toimimisesta. Toimintamalli koostuu tila–toimintopareista, jotka kuvaavat parasta mahdollista toimintoa annetussa tilassa. Oppimisen kannalta keskeistä on laskea kullekin ympäristön tilalle arvio tilan antamasta palautteesta, eli *tila-arvo* $V(s)$, jotta agentti pystyy järjestämään ympäristön tilat niiden haluttavuuden perusteella [Sutton & Barto, 1998]. Tällöin agentti voi muodostaa käsityksen ympäristön tilojen paremmuusjärjestyksestä.

Koska agentti olettaa ympäristön sisältävän stokastisia siirtymiä, eikä agentti siten voi ennakoida tulevaa ympäristön tilaa, ei agentti voi pelkkien tila-arvojen avulla valita sopivia toimintoja hakeutuakseen parempiin tiloihin. Pystyäkseen tietoisesti valitsemaan kussakin tilassa s toiminnon a, joka vie agenttia kohti parempia tiloja, on agentin tallennettava *toimintoarvoja* $Q(a,s)$, joissa tila–toimintopareille tallennetaan hyvyysarvo [Sutton & Barto, 1998]. Toimintoarvojen avulla agentti voi verrata toimintojen hyvyysarvoja keskenään vallitsevassa ympäristön tilassa ja valita näistä parhaan. Kun agentti on saanut riittävästi opetella ympäristössä toimimista, tulisi toimintoarvojen riittää ohjaamaan agentin toimintaa parhaan mahdollisen tuoton saavuttamiseksi.

Varsinainen oppiminen tapahtuu kun agentti käyttää ympäristöltä saatua palautetta päivittääkseen tila- ja toimintoarvoja. Arvojen päivittämiseen on esitetty useita eri menetelmiä, jotka voidaan jakaa *dynaamisen ohjelmoinnin* menetelmiin, *Monte Carlo* -menetelmiin ja *aikaeromenetelmiin* (*temporal difference*) [Sutton & Barto, 1998].

Dynaamisen ohjelmoinnin menetelmät vaativat täydellisen kuvauksen ympäristöstään oppiakseen optimaalisen toimintamallin. Menetelmät aloittavat suorituksen millä tahansa toimintamallilla ja tekevät malliin pieniä muutoksia. Alkuperäistä ja muokattua toimintamallia voidaan verrata laskemalla yhteen mallien tuotot kaikille ympäristön tiloille ja vertaamalla saatuja arvoja keskenään. Jos uusi toimintamalli antaa paremman tuoton, voidaan vanha malli hylätä ja etsiä yhä parempia toimintamalleja käyttämällä uutta toimintamallia lähtökohtana. Dynaamisen ohjelmoinnin menetelmät ovat siis eräänlaisia mäenkiipeämisalgoritmeja toimintamallin hakuun.

Monte Carlo -menetelmät eivät tarvitse kuvausta ympäristöstä löytääkseen optimaalisia toimintamalleja. Monte Carlo -menetelmissä agentit toimivat ympäristössä noudattaen pelkästään agentin muodostamaa toimintamallia. Kun agentti on toimillaan saanut kerättyä sopivan otoksen tila–toimintopareja ja näihin yhdistettyjä palkkioita ympäristöltä, voidaan tila–toimintoparien mukaisia toimintoarvoja päivittää tuottojen keskiarvolla. Agentin toimintamalli muodostetaan uudelleen valitsemalla kullekin tilalle se toiminto, jonka mukaan muodostettu toimintoarvo on suurin. Suorittamalla yllä kuvatut otanta- ja päivitysvaiheet riittävän monta kertaa, tulevat agentin

muodostamat toimintoarvot riittävän tarkoiksi jotta niistä voidaan muodostaa optimaalinen toimintamalli [Sutton & Barto, 1998].

Monte Carlo -oppimisalgoritmi voidaan jakaa kolmeen osaan: jakson generointiin, toimintoarvojen päivitykseen ja toimintamallin päivitykseen. Algoritmi 3 esittää Monte Carlo -oppimisalgoritmin päävaiheet [Sutton & Barto, 1998].

```
1| Alusta kaikille s ∈ S, a ∈ A(s):
2|    Q(s,a) ← satunnaisarvo ∈ ℝ
3|    π(s) ← satunnaisarvo ∈ A(s)
4|    Tuotto(s,a) ← tyhjä lista
5| Toista ikuisesti:
6|    (a) Generoi jakso noudattaen toimintamallia π
7|    (b) Jokaiselle parille (s,a) jaksossa:
8|       R ← ensimmäisen (s,a) -parin esiintymän jälkeen kertynyt tuotto
9|       Lisää R listaan Tuotto(s,a)
10|      Q(s,a) ← keskiarvo[ Tuotto(s,a) ]
11|   (c) Jokaiselle tilalle s jaksossa:
12|      π(s) ← arg max_a Q(s,a)
```

Algoritmi 3: Monte Carlo -oppimisalgoritmi.

Monikätinen rosvo -pelin kontekstissa Monte Carlo -menetelmä voisi toimia seuraavasti. Aluksi generoidaan jakso, joka koostuu ennalta määrätystä määrästä pelikertoja jaettuna mahdollisille pelikoneille (a). Koska pelikoneiden voittojakaumista ei vielä tiedetä mitään, voidaan pelikerrat jakaa tasaisesti kullekin pelikoneelle. Jakson generoinnin jälkeen ryhmitellään pelikerrat pelikoneiden mukaan ja lasketaan kunkin ryhmän tuotto, joka saadaan vähentämällä ryhmän kokonaisvoitoista pelikertojen kustannus. Uusilla ryhmien tuottoarvoilla päivitetään pelikoneiden Q-arvo, joka on jokaisen generoidun jakson tuottoarvojen keskiarvo (b). Lopuksi päivitetään agentin toimintamalli, mikä tässä kontekstissa tarkoittaa lähinnä parhaan tuoton antavan pelikoneen tunnistamista, koska ympäristön tiloja on vain yksi (c). Toimintamallin päivittämisen jälkeen voidaan jatkaa uuden jakson generoinnilla, mutta tällä kertaa pelikerrat voidaan jakaa pelikoneiden kesken älykkäämmin kun pelikerrat jaetaan pelikoneiden tuottojen odotusarvojen mukaan.

Ruudukkopelissä Monte Carlo -menetelmän ensimmäinen kierros ilmenee agentin satunnaisena liikehdintänä ruudukossa, sillä agentilla ei ole mallia ympäristöstään. Monte Carlo -menetelmä antaa agentin toimia ruudukossa niin kauan, että pelin suorituskerrat tuottavat sopivan suuren otoskoon tila–toimintopareista (a). Agentin mallin toimintoarvot päivitetään laskemalla keskiarvo kunkin tila–toimintoarvoparin tuottamasta palkkiosta (b). Lopuksi agentin toimintamalli päivitetään siten, että kullekin tilalle valitaan se toiminto, jonka toimintoarvo on suurin (c). Tämän jälkeen voidaan suorittaa uusi otantajakso (a), jolloin agentin pitäisi pystyä valitsemaan toimintoja kussakin tilassa älykkäämmin.

Kuten Monte Carlo -menetelmät, aikaeromenetelmät oppivat kokemuksen perusteella, eivätkä siten tarvitse mallia ympäristöstään. Aikaeromenetelmät eivät kuitenkaan tarvitse laajaa otosta tila–

toimintopareista ja ympäristön palautteesta päivittääkseen sisäistä malliaan, vaan toimintoarvot ja toimintamalli päivitetään jokaisen toiminnon valinnan ja palautteen vastaanottamisen yhteydessä. Agentit toimivat ympäristössään samalla tavalla kuin Monte Carlo -agentit, ja samoin kuin Monte Carlo -menetelmissä, aikaeromenetelmien toimintamalli muodostuu niistä tila–toimintopareista, joiden arvo kussakin tilassa on suurin.

Aikaeroalgoritmit käyttävät kullakin ajanhetkellä vastaanotettua palautetta $r \in \mathbb{R}$ ja opittuja toimintoarvoja $Q(s,a)$ päivittäessään toimintoarvoja. Toimintoarvojen käyttö arvojen päivityssäännössä mahdollistaa ympäristön palautteen ketjuttamisen aiemmin vierailtuihin tiloihin. Aikaeromenetelmään perustuvien algoritmien runko on esitetty algoritmissa 4 [Sutton & Barto, 1998].

```
 1| Alusta kaikille s ∈ S, a ∈ A(s):
 2|    Q(s,a) ← satunnaisarvo ∈ ℝ
 3| Toista kullekin jaksolle
 4|    Alusta s
 5|    Valitse a tilassa s Q-arvoihin perustuvalla toimintamallilla
 6|    Toista jakson jokaiselle aika-askeleelle
 7|       Suorita toiminto a, otetaan vastaan r ja s'
 8|       Valitse a' tilassa s' arvoihin Q perustuvalla toimintamallilla
 9|       Päivitä toimintoarvo Q(s,a)
10|       s ← s', a ← a'
11|    kunnes s on lopetustila
```

Algoritmi 4. Aikaeromenetelmään perustuvan oppimisalgoritmin runko.

Keskeisin Monte Carlo- ja aikaeromenetelmien ero on toiminnan suorittamisen ja toimintamallin päivityksen jaksottamisessa. Kun Monte Carlo -menetelmät jakavat nämä vaiheet toisistaan erillisiksi vaiheiksi, aikaeromenetelmissä vaiheet on yhdistetty yhdeksi vaiheeksi. Monikätinen rosvo -esimerkissä tämä tarkoittaa sitä, että toimintamalli päivitetään aina sen jälkeen, kun pelikonetta on pelattu ja sen antamat mahdolliset voitot kerätty. Ruudukkopelissä aikaeromenetelmää noudattava agentti pystyy päivittämään tilakohtaisen toimintoarvon sen jälkeen kun kyseisessä tilassa on suoritettu valittu toiminto ja ympäristön palkkio on vastaanotettu. Monte Carlo -menetelmää noudattava pelaaja sen sijaan päivittää toimintamallinsa vasta ennalta määrätyn pelimäärän jälkeen sekä monikätinen rosvo- että ruudukkopelissä. Aikaeromenetelmiä noudattavan pelaajan voidaan siis ajatella käyttävän havaintojaan tehokkaammin, koska jokainen havainto johtaa toimintamallin välittömään päivittämiseen.

Algoritmin 4 rivillä 9 agentin toimintoarvojen päivityksessä käytettyä päivityssääntöä muokkaamalla algoritmi saadaan toimimaan eri tavalla. SARSA-päivityssääntö perustuu niihin tiloihin, toimintoihin ja palautteisiin, joita agentti havaitsee toimintansa aikana. Tämän vuoksi SARSA-päivityssääntö on yksinkertainen toteuttaa ja uusi toimintoarvo voidaan laskea nopeasti. SARSA-päivityssääntö on kuvattu kaavassa 1.

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$$

Kaava 1. SARSA-päivityssääntö

Yllä esitetyssä kaavassa muuttuja $s \in S$ vastaa ympäristön tilaa ja $a \in A$ siinä suoritettua toimintoa. Muuttuja $r \in \mathbb{R}$ vastaa ympäristöltä saatua palautetta, $s' \in S$ uutta ympäristön tilaa ja $a' \in A$ uudessa tilassa suoritettua toimintoa. Päivityssäännöllä on myös parametrit α ja γ. Alennusparametri α määrää kuinka paljon päivityssääntö vaikuttaa vanhaan toimintoarvoon. Parametrin arvoväli on]0,1], missä pienemmät arvot saavat aikaan pienempiä muutoksia ja suuremmat arvot voimakkaampia muutoksia toimintoarvoihin. Jos alennusparametrin arvo on suurempi kuin 1, ei toimintamallin supistuminen optimaaliseksi toimintamalliksi ole enää taattu [Jaakkola *et al.*, 1994].

Tulevaisuudenodotusparametri γ ilmaisee kuinka suuri vaikutus tulevien tilojen arvolla $Q(s',a')$ on nykyisen tilan uuteen arvoon. Myös tulevaisuudenodotusparametrin arvoväli on]0,1].

Q-päivityssääntöä voidaan pitää SARSA-päivityssääntöä optimistisempana, sillä uuden arvon päättelyssä käytetään toimintoarvoa $a^* \in A$, joka maksimoi ympäristöltä saatavan palautteen odotusarvon. Ympäristöltä saatavan palautteen odotusarvo saadaan suoraan opituista toiminto-arvoista *Q*. Tämä johtaa siihen, että agentti oppii optimaalisen toimintamallin nopeammin kuin SARSA-päivityssääntöä käyttävä agentti, vaikka agentti ei valitsisikaan aina optimaalisia toimintoja. *Q*-päivityssääntö on kuvattu kaavassa 2.

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a^*} Q(s',a^*) - Q(s,a)]$$

Kaava 2. *Q*-päivityssääntö

Q-päivityssäännön yhteydessä on tärkeää huomata, että säännössä käytetty toiminto a^* ei välttämättä ole seuraava suoritettavaksi valittu toiminto a'. Päivityssääntö siis olettaa agentin tekevän optimaalisia valintoja kunkin aika-askeleen jälkeen, mutta agentti valitsee seuraavan suoritettavan toiminnon päivityssäännöstä erillisen toiminnonvalintamenetelmän mukaan.

Vahvistusoppimisen avulla pystytään toteuttamaan järjestelmiä, jotka ratkaisevat monimutkaisia ja vaikeaselkoisia tehtäviä. Spurlock ja Souvenir [2016] toteuttivat kokeellisen monikamera-järjestelmän, joka valitsi usean kameran lähettämästä kuvadatasta yhden näytettäväksi ruudulla perustuen kameroiden kuvaamien ihmishahmojen asentoon ja kuvakulmaan. Järjestelmä pystyi valitsemaan kameran, jonka välittämästä kuvasta tuli selvimmin ilmi hahmon toiminta. Tutkijat toteavat järjestelmän tarjoavan mahdollisuuksia turvallisuusjärjestelmille, urheiluanalytiikkalle ja älykotisovelluksille.

Ferretti tutkimusryhmineen [2016] toteutti internetselaimen lisäosan, joka oppi käyttäjien erityistarpeet selainnäkymän selkeyttämiseksi. Selainnäkymää pyrittiin selkeyttämään muokkaa-malla tekstin kirjasinta, tekstin kokoa, tausta- ja kirjasinväriä, tekstin kohdistusta ja kappale-, sana- ja kirjainvälejä. Ohjelma pystyi oppimaan käyttäjänsä tarpeiden mukaiset muokkaukset ja käyttäjien antama palaute osoitti, että ohjelma tuotti todellista hyötyä käyttäjilleen.

Lee tutkimusryhmineen [2016] käytti vahvistusoppimista kognitiivisen hätäradioverkon toteuttamiseen. Järjestelmän oppiva osa hallinnoi kognitiivisten radioiden asetuksia siten, että

hätäyksiköiden kommunikaation laatu oli järjestelmän prioriteetti. Tämän jälkeen muille käyttäjille pyrittiin takaamaan mahdollisimman laadukas kommunikaatiokanava.

Vahvistusoppimisen keskeisin ongelma liittyy opittavien toimintoarvojen määrään, joka kasvaa helposti valtavaksi käytännön sovelluksissa. Toimintoarvojen kokonaismäärä saadaan ympäristön tilojen ja agentin toimintojen karteesisesta tulosta $S \times A$. Ympäristön tila on usein kuvattu usean muuttujan avulla, jolloin ympäristön tilojen määrä saadaan kunkin muuttujan arvoavaruuden karteesisena tulona. Jos yhdenkin ympäristön tilaa kuvaavan muuttujan arvo-avaruus on kuvattu jatkuvaan arvoväliin tai on rajoiltaan ääretön, on ympäristön tilojen määrä ääretön. Naiivissa toteutuksessa toimintoarvot tallennetaan avain–arvopareina, jolloin jokainen toimintoarvo on tallennettu erikseen. Tällöin toimintoarvojen määrä tuottaa pahimmassa tapauksessa äärettömän muistitarpeen. Tästä syystä toimintoarvojen tallentamisessa on käytetty erilaisia funktion approksimointimenetelmiä, joiden avulla muistivaatimusta on saatu pienennettyä.

5.2. Toiminnonvalintamenetelmät

Yksi Monte Carlo- ja aikaeromenetelmien haasteista on tasapainoilu uusien toimintojen tutkimisen (*exploration*) ja opitun toimintamallin hyödyntämisen (*exploitation*) välillä. Parhaan toimintamallin oppimisen kannalta on edullista painottaa uusien toimintojen tutkimista, jolloin agentti oppii toimintaympäristön lainalaisuudet tarkasti niin hyvien kuin huonojenkin toimintatapojen osalta. Kun hyvät ja huonot toimintatavat voidaan erottaa toisistaan, edellyttää optimaalisen toimintamallin seuraaminen vain edullisimpien toimintoarvojen valitsemisen kussakin havaitussa tilassa. Tällöin siis painotetaan opitun toimintamallin hyödyntämistä uusien toimintojen tutkimisen kustannuksella. Ristiriita syntyy siis tarkan toimintamallin muodostamisen ja maksimaalisen tuoton saavuttamisen yhtäaikaisista tavoitteista.

Eräs yksinkertaisimmista ratkaisuista uuden tutkimisen ja opitun tiedon hyväksikäytön väliseen painotusongelmaan on niin sanottu ε-ahne valintamenetelmä [Sutton & Barto, 1998]. Agentille on ennalta määritetty pieni todennäköisyys ε, jolla agentti valitsee satunnaisen toiminnon parhaan toimintoarvon mukaisen toiminnon sijaan. Tällöin agentti pyrkii pääosin valitsemaan toimintoja, jotka takaavat parhaan mahdollisen tuoton opitun toimintamallin mukaan, mutta toisinaan agentti kokeilee toimintamalliin kuulumattomia toimintoja, jotta agentti muodostaisi mahdollisimman tarkan kuvan koko toimintaympäristöstään.

Yksinkertainen ε-ahne valintamenetelmä on kuitenkin ongelmallinen agentin toiminnan alussa ja pitkittyneen toiminnan jälkeen. Agentin toiminnan alussa toimintamalli on täynnä satunnaisia toimintoarvoja, minkä vuoksi kunkin toiminnon kokeileminen kussakin tilassa on toivottua. Tällöin olisi suotuisaa valita suuri ε-arvo uusien toimintojen tutkimisen painottamiseksi. Toisaalta agentin toiminnan jatkuessa toimintoarvot tarkentuvat ja hyvät toiminnot erottuvat selkeämmin huonoista. Tällöin opitun toimintamallin hyväksikäyttö on edullista, minkä vuoksi pienemmät ε-arvot ovat toivottuja.

Yksinkertaisen ε-ahne valintamenetelmän myötä nousee esiin uusi ongelma: sopivan ε-arvon määrittäminen. Eräs keino saavuttaa sekä suuren että pienen ε-arvon hyödyt on aloittaa suurella

arvolla ja pienentää sitä ajan kuluessa. Tällöin ongelma tosin muuttuu sopivan ε-arvon asettamisesta sopivan ε-arvon muutosnopeuden määrittämiseksi.

Toiminnonvalintaongelmaan on esitetty hienostuneempia menetelmiä, jotka pyrkivät tilakohtaisesti päättelemään sopivan todennäköisyyden kokeilevan toiminnon valitsemiseksi. Xia ja Zhao [2016] esittävät Bayesin teoreemaan pohjautuvan toiminnonvalintamenetelmän, jonka he osoittavat toimivan erityisen hyvin sovelluksissa, joissa agentin toimintaympäristön tilat ovat luonteeltaan jatkuvia. Auerin tutkimusryhmän [2002] kehittämän UCB -toiminnonvalinta-menetelmä pyrkii samaan aikaan maksimoimaan tulevan palkkion ja kokeilemaan toimintoja, joita on kokeiltu harvemmin monikätinen rosvo -pelissä. Kaava 3 esittää UCB-menetelmän toiminnanvalintasäännön. Kaavassa $\overline{x_j}$ on vaihtoehdon j odotusarvo, n_j on vaihtoehdon j suoritusmäärä ja n kaikkien vaihtoehtojen suoritusmäärien summa.

$$UCB = \overline{x_j} + \sqrt{\frac{2 \ln n}{n_j}}$$

Kaava 3. UCB-algoritmi valitsee vaihtoehdon j, joka maksimoi UCB-arvon.

UCB-menetelmän laskentakaava on kaksiosainen: se arvottaa kunkin toiminnon sen tuottaman palkkion keskiarvon (1) ja kyseisen toiminnon kokeilujen määrän ja suoritettujen toimintojen kokonaismäärän suhdeluvun perusteella (2). Tällöin siis kutakin toimintoa suoritetaan yhtä monta kertaa, jos ne tuottavat keskimäärin yhtä suuren palkkion. Palkkiojakauman ollessa epätasainen, UCB-menetelmä valitsee useammin toimintoja, jotka tuottavat keskimäärin suurimman palkkion. Valintojen kokonaismäärän kasvaessa toiminnon kokeilujen määrän ja suoritettujen toimintojen kokonaismäärän suhdeluvun painoarvo pienenee, minkä johdosta valintamenetelmä alkaa tehdä ahneempia valintoja.

Kocsis ja Szepesvári [2006] esittävät UCB-menetelmään pohjautuvan UCT-toiminnon-valintamenetelmän (kaava 4), joka on sovellettavissa pelipuiden yhteydessä. Kuten UCB-menetelmä, UCT-menetelmä valitsee toimintoja sekä niiden antaman tuoton että toiminnon suoritusmäärän mukaan. UCT pystyy ottamaan huomioon myös ympäristön vallitsevan tilan valitessaan seuraavaa toimintoa.

$$UCT = \frac{w_i}{n_i} + c\sqrt{\frac{\ln t}{n_i}}$$

Kaava 4. UCT-menetelmä valitsee vaihtoehdon i, joka maksimoi UCT-arvon.

UCT-arvon laskentakaavassa w_i tarkoittaa vaihtoehdon i tuottamaa voitettujen pelien määrää ja n_i sitä kuinka monta kertaa valinta i on tehty. Kaikkien tällä ajanhetkellä valittavissa olevien vaihtoehtojen valintakertojen määrä merkitään arvolla t. Kaavan parametrilla c voidaan säätää kuinka kokeilunhaluinen menetelmä on.

5.3. Graafihakualgoritmien ja vahvistusoppimisen yhteys

Vaikka koneoppimista on onnistuneesti sovellettu graafihaun ohjaamiseen [Arfaee *et al*., 2011; Fink, 2007; Petrik & Zilberstein, 2008], tarjoaa vahvistusoppiminen houkuttelevan mahdollisuuden parantaa koneoppivan graafihaun sovellettavuutta. Sen sijaan että graafin solmujen piirteitä tarvitsisi hakea, koostaa tai käsitellä, vahvistusoppimiseen perustuva graafihaku voi toimia myös solmukohtaisten piirteiden puuttuessa. Käytännössä ainoa asia, jota solmulta vaaditaan vahvistusoppimiseen perustuvassa graafihaussa on, että se voidaan erottaa muista solmuista esimerkiksi yksilöivän tunnisteen avulla. Tämä tarkoittaa, että vahvistusoppimiseen perustuva graafihaku tarvitsee yhtä paljon tietoa graafin solmuista kuin heikot hakualgoritmit, mikä asettaa graafin kuvaukseen liittyvät vaatimukset pienemmiksi kuin mitä heuristiset hakumenetelmät vaativat. Vahvistusoppimista pohditaan seuraavaksi graafihaussa. Merkintätapojen sekaannuksen välttämiseksi graafin solmuihin viitataan merkinnällä V_G ja vahvistusoppimisessa käytettyihin tila-arvoihin merkinnällä V_{RL}.

Vahvistusoppimisessa agentti ylläpitää tila- ja toimintoarvoja, joiden avulla muodostetaan agentin toimintamalli. Tila- ja toimintoarvojen päivitysmenetelmien nähdään sisältävän kaiken tarpeellisen informaation, joka kokemuksen kautta on opittu. Agentin oletetaan toimivan aina oppimansa toimintamallin perusteella, mikä ajoittain tarkoittaa myös sitä, että agentti voi tehdä huonojakin valintoja tiloissa, joiden ominaisuudet tunnetaan hyvin.

Toteutettaessa vahvistusoppimiseen perustuvaa graafihakualgoritmia, hakualgoritmien tulee hylätä graafista havaittavissa oleva informaatio ja noudattaa erillistä toimintamallia, jota voidaan päivittää hakualgoritmin toiminnan perusteella. Graafin solmut V_G toimivat hakutehtävän tiloina S ja solmusta poispäin kulkevat kaaret $E(v,v')$ toimivat tilakohtaisina toimintoina $A(s)$, missä $v, v' \in V_G$ ja $s \in S$. Graafin kuvauksesta saadaan siis muodostettua ympäristö ja sen lainalaisuudet asettamalla $S = V_G$ ja $A(s) = E(v,v')$. Tämän perusteella oppivan järjestelmän tila-arvojen V_{RL} parametrina toimivat graafin solmut, jolloin tila-arvot kuvataan joukkona $V_{RL}(v)$, $v \in V_G$. Samoin toimintoarvojen molemmat parametrit saadaan graafin solmuista; toimintoarvojen ensimmäinen parametri on ympäristön tila, eli graafin solmu $v \in V_G$ ja toinen parametri agentille sallittu toiminto kyseisessä tilassa, eli solmusta poispäin kulkeva kaari $e \in E(v,v')$, jolloin toimintoarvot ovat muotoa $Q(v,e)$. Lähtösolmun esittäminen erikseen toimintoarvon parametrina on turhaa, sillä lähtösolmu on saatavissa kaaresta e. Toimintoarvojen esittämiseen tarvitaan siis vain kaaren parametrit, jolloin toimintoarvot ovat muotoa $Q(v,v')$.

Vahvistusoppimisen ja hakualgoritmien toiminnassa on yksi keskeinen ero. Vahvistus-oppimisessa jokainen toiminto on peruuttamaton; kun ympäristön tila on kerran vaihtunut, ei agentti voi perua edellisiä toimintojaan ja palata aiempaan ympäristön tilaan. Sen sijaan agentin on valittava toimintoja, jotka vievät agentin takaisin aiempaan tilaan, jos mahdollista. Tämä on perusteltua siksi, että agentti oppii välttämään ylimääräisiä toimintosarjoja joutuessaan jakamaan saavuttamaansa tuottoa ylimääräisille solmuille. Aiemmin käsitellyt graafihakumenetelmät ovat kuitenkin sallineet haun laajentamisen aiemmista tiloista, jolloin paras mahdollinen ratkaisu on voitu löytää siitä huolimatta, että algoritmi on tutkinut ylimääräisiä solmuja.

Sopivan toimintamallin oppiminen puhtaasti vahvistusoppimisen mukaisesti on mahdollista, mutta tällöin on epätodennäköistä, että algoritmi löytää parhaita mahdollisia ratkaisuja. Tämä johtaa siihen, että oppiminen pitää suorittaa ennen algoritmin käyttöä todellisen hakuongelman ratkaisussa erillisen oppimisvaiheen puitteissa. Jos algoritmin halutaan oppivan todellisten hakuongelmien aikana ja samalla löytää parhaita mahdollisia ratkaisuja, on vahvistusoppimisen vaatimusta peruuttamattomista toiminnoista muokattava. Tällöin hakualgoritmi voi laajentaa aiemmista tiloista tutkimattomiin avoimiin solmuihin, kuten muissakin hakualgoritmeissa. Kun ratkaisu on löydetty tai muu lopetusehto täyttyy, voidaan ratkaisuista ja muista avoimista solmuista koostaa polku aloitussolmuun ja käyttää saatua polkua päivittämään agentin sisäistä mallia. Toimintoarvojen päivittämisessä käytetyt aika-askeleet määritetään vasta sitten, kun koko polku aloitussolmusta kohdesolmuun on määritetty ja polun sisältämien solmujen voidaan nähdä laajentuneen ajallisesti samassa järjestyksessä kuin ne ratkaisupolussa sijaitsevat riippumatta siitä mitä ylimääräisiä askelia haku oli suorittanut.

6. Solmujen välisten etäisyyksien oppiminen

Tässä luvussa kuvataan toteuttamani oppivan hakumenetelmän keskeisimmät osat. Koska hakumenetelmä toimii ahneen hakumenetelmän tavoin sillä erotuksella, että solmujen ominaisuuksien sijaan oppiva haku käyttää sisäistä malliaan solmujen järjestämiseen, keskitytään hakumenetelmän kuvauksessa sisäisen mallin päivitysmenetelmiin. Luvun lopussa esitetään muokattu UCT-toiminnonvalintamenetelmä, joka käyttää haluttavuusarvojen sijaan etäisyysarvoja toimintoja arvottaessaan. Muokattu UCT-menetelmä soveltuu paremmin graafihaun ohjaukseen ja on osa toteuttamaani oppivaa graafihakumenetelmää.

Solmujen välisen polun pituuden käyttäminen avointen solmujen listan järjestämiseen on ideaali, jota heuristisilla hauilla pyritään jäljittämään. Yksittäisiin solmuihin ei kuitenkaan sisälly tietoa, jonka avulla voitaisiin päätellä solmusta lähtevän ja toiseen solmuun päättyvän polun pituus. Heuristisen haun soveltuvuusfunktiot pyrkivät approksimoimaan solmun ja kohdesolmun välistä etäisyyttä. Approksimointiin liittyvän epätarkkuuden ja epätäydellisyyden vuoksi on kuitenkin mahdollista – ja todennäköistä – että heuristiset hakualgoritmit valitsevat välillä ratkaisupolkuun kuulumattomia solmuja.

Soveltamalla Monte Carlo -oppimismenetelmää, on hakualgoritmin mahdollista oppia arvottamaan solmuparien välisiä etäisyyksiä niiden välillä kulkevan lyhimmän polun mukaan. Algoritmin ei välttämättä tarvitse oppia solmujen välisten polkujen todellisia arvoja, vaan riittää että solmupareille annettavat arvot ovat vertailtavissa, jotta hakua voidaan ohjata lyhimmän hakupolun mukaan. Oppivan haun tehtävänä on siis oppia kuvaus $f(n',n) \rightarrow \mathbb{R}$, missä n' on käsiteltävä solmu ja n on tunnettu kohdesolmu. Solmujen n' ja n välisen lyhimmän polun todellinen pituus $d(n',n)$ kuuluu epänegatiivisten lukujen joukkoon $\mathbb{N}_0$, joka on järjestetty joukko. Opittavien etäisyysarvojen vertailtavuusehdon täyttämiseksi on kuvauksen f arvojen oltava keskenään verrattavissa ja vertailujen tuotettava sama vertailutulos niitä vastaavien todellisten etäisyysarvojen kanssa:

- $f(n',n) > f(n'',n)$, kun $d(n',n) > d(n'',n)$
- $f(n',n) < f(n'',n)$, kun $d(n',n) < d(n'',n)$
- $f(n',n) = f(n'',n)$, kun $d(n',n) = d(n'',n)$

Algoritmin opetusvaiheessa on mahdollista, että haku ei löydä kohdesolmua annetussa aika- ja askelrajassa. Jotta algoritmi välttää polkuja, joista ei ole löytynyt ratkaisua hakutehtävään, on etäisyysarvoja polun solmujen ja kohdesolmun välillä kasvatettava, jolloin haku ohjautuu kohti tutkimattomia hakupolkuja. Tämä vastaa rangaistusta, eli negatiivista palautetta agentin toiminnasta. Koska epäonnistuneen hakupolun solmujen etäisyyttä kohdesolmusta ei voida päätellä, on luontevaa käyttää etäisyysarvovälin maksimiarvoa etäisyysarvon päivittämiseen. Todellisten etäisyyksien funktion $d(n',n)$ arvoväli on kuitenkin $[0,\infty[$, minkä vuoksi todellisen etäisyyden maksimiarvoa ei voida käyttää etäisyysarvon päivittämiseen. Sopiva maksimiarvo saadaan rajaamalla kuvauksen f arvoväliä siten, että kuvauksen vertailuehdot silti täyttyvät. Tässä tutkielmassa kuvauksen f arvoväli rajataan arvoihin $[0,1[$ kaavan 5 mukaan, jossa f-arvo lasketaan havaitusta todellisesta etäisyysarvosta d.

$$f(d) = \frac{d}{d+1}$$

Kaava 5. Etäisyysarvon laskentakaava, jolla havaittu etäisyys muutetaan arvovälille $[0,1[$.

Oppiva hakualgoritmi generoi hakupuun, jonka avulla solmujen välisiä etäisyyksiä voidaan päivittää. Hakupuu alkaa lähtösolmusta ja sen lehtisolmuihin kuuluu kohdesolmu, jos ratkaisu hakutehtävään löydettiin. Muut lehtisolmut ovat niiden hakupolkujen päitä, jotka eivät päättyneet kohdesolmuun. Generoitu hakupuu tarjoaa mahdollisuuden päivittää useiden erilaisten solmuparien välistä etäisyysarvoa f:

1. hakupuun yhden haaran sisältämien solmujen väliset etäisyydet
2. etäisyys onnistuneen hakupolun solmujen ja kohdesolmun välillä
3. etäisyys epäonnistuneiden hakupolkujen solmujen ja kohdesolmun välillä

Hakupolun sisällä kaikkien solmujen välinen etäisyys (1.) lasketaan hakupolusta siten, että solmujen välinen etäisyys on solmujen paikkaindeksien erotuksen itseisarvo. Onnistuneen hakupolun solmujen etäisyys kohdesolmusta (2.) on kohdan (1.) triviaalitapaus, jossa etäisyys lasketaan samalla tavalla solmun ja kohdesolmun paikkaindeksien mukaan. Epäonnistuneiden polkujen solmujen ja kohdesolmun etäisyyden (3.) laskeminen ei kuitenkaan ole niin suoraviivaista. Jos haku onnistui ja hakupuuhun kuuluu onnistunut hakupolku, voidaan epäonnistuneissa hakupoluissa kulkea taaksepäin kunnes päädytään onnistuneen hakupolun solmuun. Epäonnistuneen hakupolun solmun ja kohdesolmun välinen etäisyys voidaan laskea takaisinpäin kuljettujen solmujen määrän ja onnistuneen polun solmun ja kohdesolmun välisen etäisyyden summana. Jos haku epäonnistui ja yhtäkään onnistunutta hakupolkua ei kuulu hakupuuhun, käytetään etäisyysarvovälin ylärajaa etäisyysarvon päivittämiseen.

Hakupolkujen kaikkien solmujen välisten etäisyyksien (1.) ja onnistuneen hakupolun solmujen ja kohdesolmun välisen etäisyyden (2.) päivitysmenetelmä on kuvattu algoritmissa 5.

```
1| Toista kaikille s ∈ polku
2|    Toista kaikille s' ∈ polku
3|       etäisyys ← |s - s'|
4|       Päivitä((s,s'),etäisyys)
```

Algoritmi 5. Hakupolkujen solmujen välisten etäisyysarvojen päivitysmenetelmä.

Epäonnistuneen hakupolun solmujen ja kohdesolmun välisten etäisyysarvojen päivitysmenetelmä (3.) on esitetty algoritmissa 6.

```
1| Olkoon polku onnistunut hakupolku ja polku' epäonnistunut hakupolku
2| Toista kaikille s ∈ polku'
3|    s* ← s
4|    Toista
5|       s* ← s*.vanhempi
6|    kunnes s* ∈ polku tai s* = ∅
7|    if s* ≠ ∅
8|       etäisyys ← |s - s*| + |s* - s^kohde|
9|    else
10|      etäisyys ← 1
11|   end if
12|   Päivitä((s,s'),etäisyys)
```

Algoritmi 6. Epäonnistuneen hakupolun solmujen ja kohdesolmun välisen etäisyyden päivitysmenetelmä.

Algoritmissa 5 ja 6 solmujen välinen havaittu etäisyys – hakupolusta saatujen paikkaindeksien erotus – esitetään merkinnällä $|s - s'|$. Algoritmien käyttämä Päivitä-menetelmä on esitetty algoritmissa 7, jossa solmuparille (s,s') opittu etäisyysarvo esitetään merkinnällä $f_{s,s'}$ ja solmuparin (s,s') valintakertojen määrä esitetään merkinnällä $c_{s,s'}$.

```
1| Olkoon alennus pieni vakio arvoväliltä [0,1]
2| Päivitä((s,s'),etäisyys)
3|    f ← etäisyys / (etäisyys + 1)
4|    if f > f_{s,s'}
5|       a ← alennus
6|    else
7|       a ← 1
8|    end if
9|    f_{s,s'} ← [(f_{s,s'} × c_{s,s'}) + (f × a)] / (c_{s,s'} + a)
10|   c_{s,s'} ← c_{s,s'} + a
```

Algoritmi 7. Kuvauksen *f* arvojen päivittämiseen käytetty laskentamenetelmä.

Epäonnistuneen hakupolun solmujen päivittämiseen käytettävä arvo on mahdollisesti korkeampi kuin kyseisen solmun todellista etäisyyttä vastaava etäisyysarvo. Samoin epäonnistuneessa haussa käytettävä maksimiarvo muokkaa etäisyysarvoa vahvasti. Todellisuutta suurempien arvojen käyttäminen etäisyysarvojen päivittämiseen voi johtaa siihen, että algoritmi suosii löytämäänsä ratkaisua vaikkei se olisikaan paras mahdollinen polku. Tämän välttämiseksi algoritmi 7 käyttää alennettua painoarvoa päivittäessään etäisyysarvoaan opittua etäisyysarvoa suuremmilla arvoilla. Suurempien arvojen alentaminen jättää mahdollisuuden pienten etäisyysarvojen poisoppimiseen, mikä on tärkeää tilanteissa, joissa graafi saattaa muuttua ja opitut arvot lakkaavat vastaamasta todellisuutta. Pienempien arvojen suosiminen kuitenkin rohkaisee hakua ohjautumaan solmuihin, joiden kautta ollaan aiemmin löydetty lupaavia hakutuloksia.

Laadukkaan oppimisen takaamiseksi haun on tasapainoteltava opittujen etäisyysarvojen hyödyntämisen ja uusien polkujen kokeilemisen välillä. Oppimisen hyödyntämisen ja uuden kokeilemisen välillä tasapainotteluun käytetään UCT-menetelmää [Kocsis & Szepesvári, 2006] mukailevaa laskentakaavaa. Koska UCT-menetelmässä kokemusta hyödyntävä osuus perustuu onnistumisten määrään, mikä ei suoraan palvele oppivaa hakumenetelmää, tulee muokatun UCT-laskentakaavan perustua opittuihin etäisyysarvoihin. Opituissa etäisyysarvoissa parempia vaihtoehtoja kuvaavat pienemmät arvot, jotka kertovat lyhyemmästä matkasta kohdesolmuun, kun taas UCT-menetelmässä suuremmat arvot kuvasivat suurempaa onnistumisprosenttia ja olivat siten pienempiä arvoja suotuisampia. Tämän vuoksi etäisyyksiin perustuvassa laskentamenetelmässä suositaan pienempiä UCT-arvoja. Oppivaa hakua ohjaava, uusia vaihtoehtoja koettava osuus saadaan suoraan UCT-menetelmästä vanhempi–lapsisolmujen valintamäärien mukaan sillä erotuksella, että pienempien UCT-arvojen suosimisen vuoksi tämän osan vastaluku lasketaan UCT-arvoon mukaan. Oppivalle haulle soveltuva, muokattu UCT-laskentakaava on esitetty kaavassa 6.

$$UCT(s,s') = D(s,s') - c\sqrt{\frac{\ln n_{s^p,s'}}{n_{s,s'}}}$$

Kaava 6. Muokattu UCT-laskentakaava, joka käsittelee onnistumisprosentin sijaan etäisyysarvoa.

Kaavan 6 mukaan lasketaan UCT-arvo solmujen s ja s' välille. $D(s,s')$ merkitsee agentin oppimaa etäisyysarvoa solmujen s ja s' välille. Arvo $n_{s,s'}$ osoittaa montako kertaa solmu s on laajennettu silloin kun solmu s' on ollut haun kohdesolmu. Merkintä s^p tarkoittaa solmun s vanhempaa.

Oppivan hakualgoritmin toiminta jakautuu kahteen osaan: ratkaisun hakuun ja etäisyysarvojen päivittämiseen. Ensimmäinen vaihe toimii kuten ahne paras ensin haku. Solmuihin liitetään soveltuvuusarvo, joka saadaan kaavan 6 mukaisella mukautetulla UCT-laskentakaavalla. Avoimien solmujen listalta valitaan laajennettavaksi solmu, jonka UCT-arvo on pienin ja haku jatkuu kunnes ratkaisu löydetään tai jokin ennalta määrätty aikaraja täyttyy. Hakuvaiheen päätyttyä hakualgoritmi on tuottanut hakupuun, jossa saattaa olla onnistunut, kohdesolmuun johtava hakupolku. Tätä hakupuuta käytetään etäisyysarvojen päivitysvaiheessa. Oppivan hakualgoritmin toinen vaihe päivittää hakupuun jokaisen haaran solmujen väliset etäisyydet ja jokaisen solmun ja kohdesolmun välisen etäisyyden algoritmien 5 ja 6 mukaan.

7. Oppivan haun toiminnan mittaaminen

Heikkojen- ja heuristisen hakumenetelmien tehokkuuden vertaaminen toisiinsa voidaan toteuttaa osittain tutkimalla algoritmin kuvausta ja laskemalla hakujen suorittamien toimenpiteiden määriä esimerkkitapauksilla, sillä algoritmien deterministisen luonteen vuoksi algoritmin kulku on pääteltävissä. Stokastisten hakumenetelmien yhteydessä tällaista tehokkuuden päättelyä ja vertailua on kuitenkin hankala toteuttaa. Oppivan graafihakualgoritmin tehokkuuden mittaamiseksi suoritetaan kaksi koetta, joissa oppivalle haulle esitetään laajempia hakutehtäviä.

Ensimmäisessä kokeessa oppivalle haulle esitetään useita hakutehtäviä satunnaisesti generoiduissa graafeissa. Graafit generoidaan siten, että mittaustulosten vertailukohtana voidaan käyttää A*-hakumenetelmän vastaavia tuloksia. Kokeella pyritään selvittämään voiko oppiva haku toimia yhtä tehokkaasti kuin hyvin suunniteltu heuristinen haku ja onko oppimiseen kulunut aika kohtuullisissa rajoissa.

Toisessa kokeessa oppiva haku suoritetaan sosiaalista verkkoa kuvaavassa graafissa, jossa heuristisen haun suorittaminen ei ole mielekästä. Graafi tuotetaan tosielämän sosiaalista verkkoa kuvaavasta datasta [Leskovec & Krevl, 2014; McAuley & Leskovec, 2012]. Kokeella halutaan selvittää saavuttaako oppiva haku hyvän suoritustason monimutkaisessa, tosielämän tilannetta kuvaavassa graafissa ja kuinka suuren parannuksen oppiva haku tarjoaa verrattuna heikkoihin hakumenetelmiin, jotka ovat nykysovelluksissa yleisesti käytetty graafihakumenetelmä sosiaalisten verkkojen yhteydessä.

Kaikki kokeissa käytetyt hakumenetelmät ylläpitävät suljettujen solmujen listaa, jonka avulla graafin solmut käsitellään korkeintaan kerran. Tällöin solmujen toistuva käsittely ei vaikuta kokeessa käytettyihin mittareihin.

7.1. Oppivan haun mittaaminen satunnaisesti generoiduissa graafeissa

Oppivan- ja heuristisen haun tehokkuuseroja tutkitaan euklidiseen avaruuteen kuvatussa graafissa, johon toteutetaan heuristinen haku, jonka soveltuvuusfunktio arvottaa avoimet solmut solmun

etäisyytenä kohdesolmusta. Oppivan haun tulisi oppia graafin malli siten, että se löytäisi samat tai yhtä hyvät ratkaisupolut kuin heuristinen haku. Oppivalla haulla on periaatteessa mahdollisuus oppia myös hakupolkujen varrella olevat umpikujat ja pystyä välttämään niitä toisin kuin heuristinen haku, joka käy aina tällaiset umpikujat läpi, jos umpikujassa on soveltuvuusfunktion mukaan parempia solmuja.

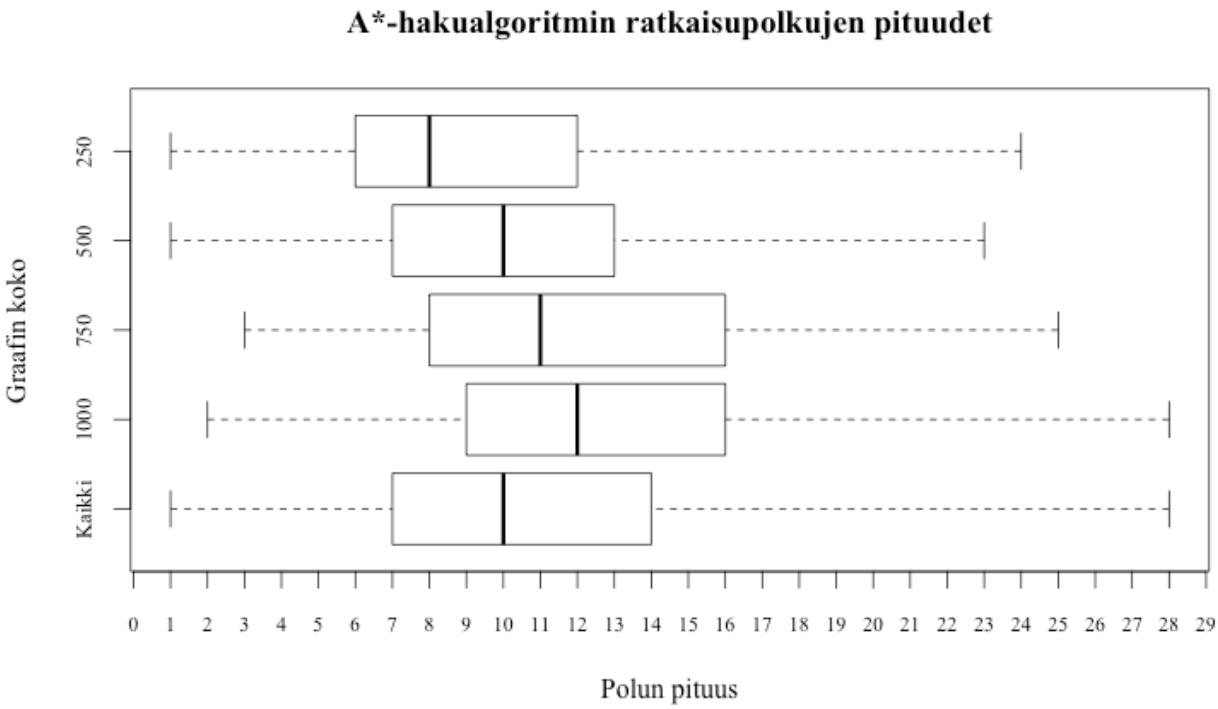
Haun tehokkuutta mitataan löydetyn ratkaisun kokonaispituuden ja haun aikana suljettujen solmujen lukumäärän avulla. Nämä mittarit arvostavat hakuja, joissa ratkaisu on mahdollisimman lyhyt ja joissa suoritetaan mahdollisimman vähän ylimääräisiä askelia. Oppimisnopeutta arvioidaan opetussyklien määränä, joka tarvitaan saavuttamaan tilanne, jonka jälkeen haun tehokkuus ei kasva merkittävästi.

Koetta varten generoidaan graafeja, joissa on 250, 500, 750 ja 1000 solmua. Kustakin kokoluokasta generoidaan 25 graafia. Graafien solmut kuvaavat pisteitä kolmiulotteisessa avaruudessa. Kutakin solmua yhdistää kolmesta seitsemään kaksisuuntaista kaarta, jotka generoidaan satunnaisesti siten, että ne yhdistävät solmuja, jotka sijaitsevat lähellä toisiaan.

Jokaiselle generoidulle graafille generoidaan 10 hakutehtävää siten, että lähtö- ja kohdesolmuiksi valitaan satunnaiset, toisistaan erilliset solmut annetusta graafista. Hakutehtävät esitetään A*-haulle, jonka toiminnasta mitataan ratkaisun pituus ja suljettujen solmujen määrä. Koska A*-haku toimii deterministisesti ja graafi pysyy muuttumattomana, riittää että A*-haku suoritetaan vain kerran kullekin hakutehtävälle. Teoriassa A*-haun löytämä ratkaisu on lyhin mahdollinen polku lähtö- ja kohdesolmun välillä ja pienin mahdollinen suljettujen solmujen määrä on sama kuin ratkaisupolun solmujen lukumäärä. Tämän vuoksi A*-haun ratkaisupolun pituutta käytetään vertailukohtana hakualgoritmien löytämien ratkaisujen pituudelle ja suljettujen solmujen määrälle.

Oppimisen kehittymistä ja oppivan haun tehokkuuden muutosta seurataan esittämällä hakutehtävät oppivalle haulle 1000 kertaa. Haun tuottaman ratkaisupolun pituus ja suljettujen solmujen määrä mitataan 10 suorituskerran välein.

Hakutehtävien satunnaisgeneroinnin vuoksi tehtävien lyhintä ratkaisua ei voida ennalta tietää. Lyhimmän ratkaisun pituuden voidaan kuitenkin olettaa vaikuttavan oppimistuloksiin, sillä pitempien polkujen päässä sijaitsevien kohdesolmujen löytämisen pitäisi olla haastavampaa, kun avointen solmujen lista kasvaa kunkin laajennettavaksi valitun solmun yhteydessä. Koska A*-haku löytää parhaan mahdollisen ratkaisun hakutehtävään, käytetään A*-haun tuottaman ratkaisupolun pituutta parhaana ratkaisuna hakutehtävään. Kuvassa 19 on esitetty hakutehtävien pituuksien jakauma laatikko–janakuviona jaoteltuna graafin koon mukaan. Laatikko–janakuvioiden keskiviivat vastaavat kuvatun joukon mediaania, laatikkojen reunat ensimmäistä ja kolmatta kvartiilia ja janojen päät joukon äärimmäisiä arvoja.



Kuva 19. Hakutehtävien parhaiden ratkaisujen pituuksien jakauma A*-hakualgoritmin mukaan.

Kuvasta 19 ilmenee kuinka A*-hakualgoritmin tuottaman ratkaisupolun pituuksien jakaumat ovat jokseenkin samanlaisia. Ratkaisupolkujen pituudet kasvavat hieman graafin koon kasvaessa, mutta kunkin graafin kokoluokan ratkaisupolkujen pituuksien mediaani kuuluu välille 8–12 ja kaikkien graafien ratkaisupolkujen mediaanipituus on 10. Eri kokoisissa graafeissa suoritettuja hakuja voidaan siis pitää vertailukelpoisina hakupolkujen pituuksien perusteella.

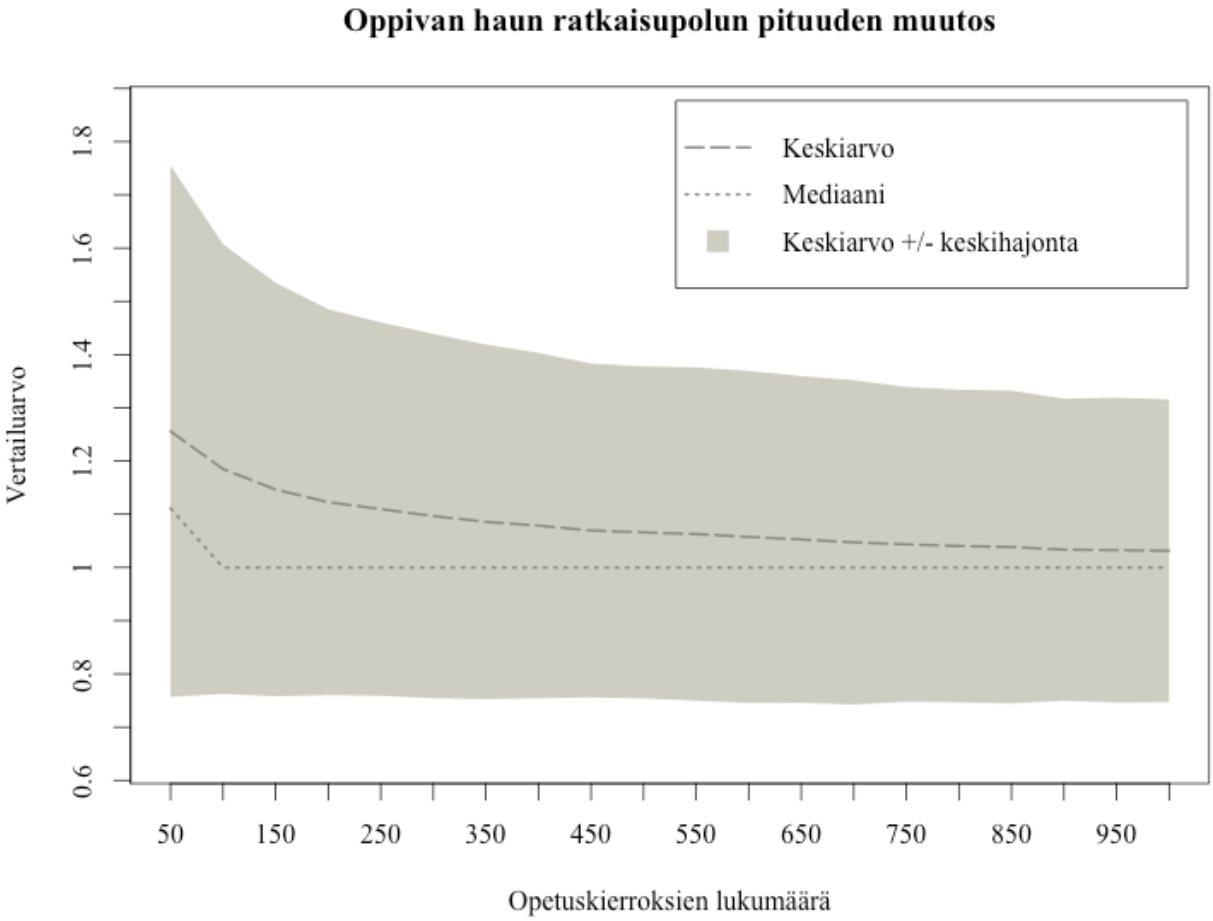
Koska hakutehtävien parhaiden mahdollisten ratkaisupolkujen pituus vaihtelee välillä 1–28, eivät ratkaisupolkujen pituudet ole suoraan verrattavissa hakutehtävien välillä. Samoin suljettujen solmujen määrät eivät ole suoraan verrattavissa, sillä pienin mahdollinen suljettujen solmujen määrä kullekin hakutehtävälle on yhtä suuri parhaan mahdollisen ratkaisupolun pituuden kanssa.

Kun suljettujen solmujen määrä on sama kuin parhaan mahdollisen ratkaisupolun pituus, on haku löytänyt parhaan mahdollisen ratkaisupolun ja kulkenut lähtösolmusta kohdesolmuun sulkematta yhtäkään parhaaseen mahdolliseen ratkaisupolkuun kuulumatonta solmua. Jotta mittaustuloksia voidaan verrata hakutehtävien välillä, käytetään vertailuun suhdelukua, joka saadaan jakamalla havaittu arvo parhaan mahdollisen ratkaisupolun pituudella. Tällöin parhaan mahdollisen ratkaisupolun pituudesta ja suljettujen pienimmästä mahdollisesta määrästä saatu vertailukelpoinen arvo on 1.

Koska vertailukelpoinen arvo lasketaan suhteessa A*-haun tuottaman ratkaisupolun pituuteen, on A*-haun tuottaman ratkaisupolun pituuden arvo aina 1. A*-haun yhteydessä onkin mielekästä laskea vertailukelpoinen arvo vain suljettujen solmujen määrälle.

Oppivan haun tehokkuus kehittyi selvästi suoritusmäärän kasvaessa. Vaikka haku ei aina löydä parasta mahdollista ratkaisua annettuun hakutehtävään, löydetyn ratkaisun pituus ja suljettujen solmujen määrä ovat mediaanin mukaan yhtä suuret A*-haun ratkaisun pituuden kanssa

opetusvaiheen päätyttyä. Kuvasta 20 ilmenee kuinka hakupolun pituuden suhde parhaan mahdollisen hakupolun pituuteen kehittyy opetusajojen määrän kasvaessa kaikkien graafien kaikissa hakutehtävissä.

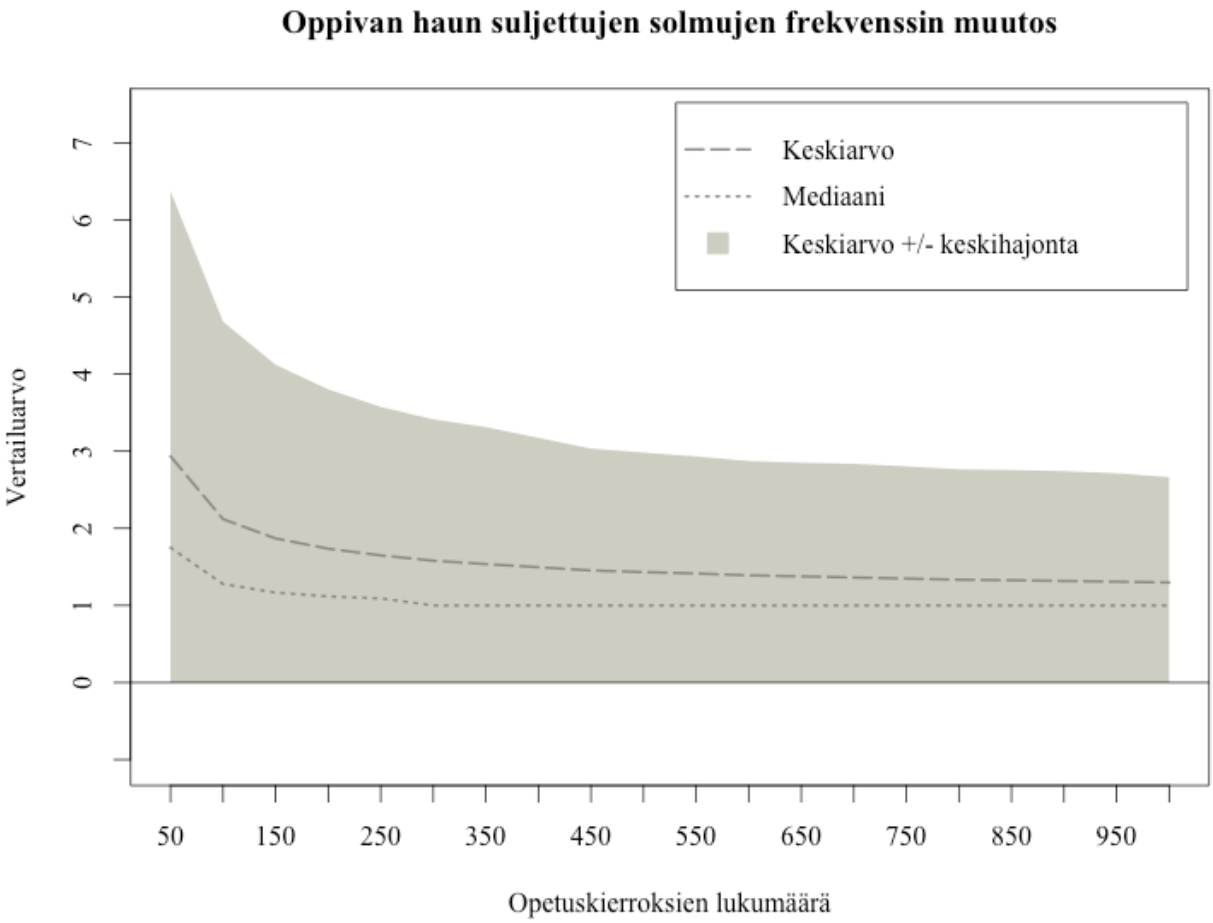


Kuva 20. Oppivan haun ratkaisupolun kehitys oppimisen edetessä kaikissa graafeissa.

Oppivan haun tuottama ratkaisu kehittyy oppimisen edetessä kohti lyhyempiä ratkaisuja. Oppivan haun ratkaisun pituuden vertailuarvo on oppivan haun alussa keskiarvoltaan 1,2560 mediaanin ollessa 1,1111. Opetusvaiheen lopussa ratkaisun pituuden vertailuarvon keskiarvo on 1,0314 ja mediaani 1,0000. Kuvaajan mukaan algoritmi pystyy muokkaamaan sisäistä malliaan siten, että sen tuottama ratkaisupolku ei muutu aiempaa huonommaksi. Myös ratkaisupolkujen pituuden hajonta pienenee oppimisen edetessä, mikä kertoo osaltaan algoritmin sisäisen mallin kehittymisestä kohti parasta mahdollista ratkaisua.

Oppivan haun tuottaman ratkaisupolun pituuden kehitys vaikuttaa olevan voimakkaimmillaan ensimmäisten 100 opetuskierroksen aikana. Vaikka ratkaisupolkujen pituuden keskiarvo ja mediaani muuttuivat hitaammin 100. opetuskierroksen jälkeen, ovat niiden arvot kuitenkin pienentyneet kaikkien opetusajojen aikana. Ratkaisupolun pituuden vertailuarvon keskiarvo 100. opetuskierroksen jälkeen on 1,1850 ja mediaani 1,0000.

Oppivan haun suljettujen solmujen määrän muutos on polun pituuden muutosta dramaattisempi. Kuva 21 esittää kaikkien graafien kaikissa hakutehtävissä havaittujen suljettujen solmujen määrän vertailuarvon opetuskierrosmäärän funktiona.



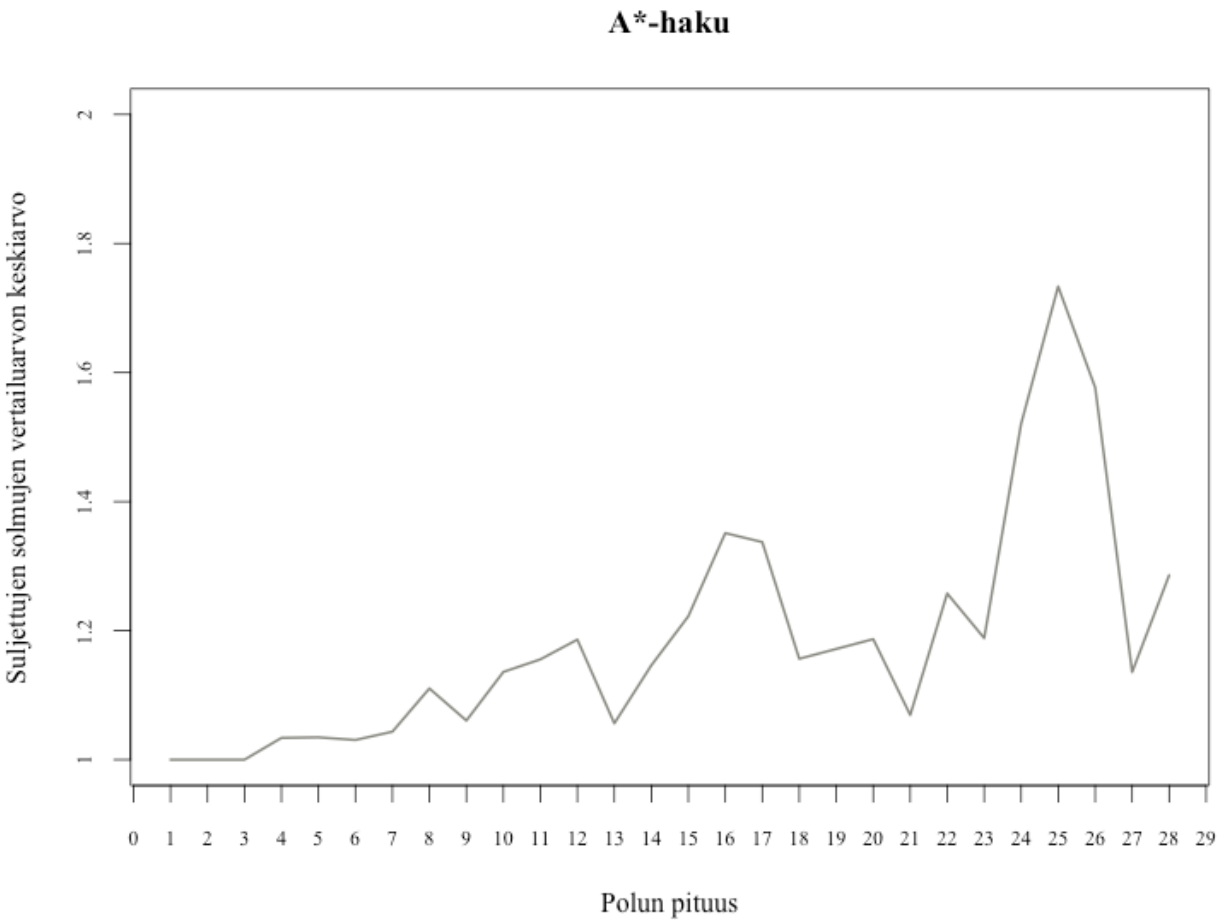
Kuva 21. Oppivan haun sulkemien solmujen määrän vertailuarvon kehitys kaikissa graafeissa.

Suljettujen solmujen määrän muutos oppimisen aikana mukailee ratkaisupolun pituuden muutosta. Suljettujen solmujen määrän vertailuarvon keskiarvo on opetusvaiheen alussa 2,9351 ja lopussa 1,2975. Vertailuarvon mediaani alkaa arvosta 1,7500 ja päättyy arvoon 1,0000. Suljettujen solmujen määrä laskee nopeasti ensimmäisten 100 opetuskierroksen aikana. Tämän jälkeen suljettujen solmujen määrä jatkaa pienenemistään, mutta muutosnopeus hidastuu huomattavasti. Vertailuarvon keskiarvo 100. opetuskierroksen jälkeen on 2,1191 ja mediaani 1,2778.

A*-haun vertailuarvon keskiarvo suljettujen solmujen määrälle kaikissa graafeissa on 1,1239 ja mediaani 1,0000. Vertailuarvojen mediaanin mukaan haut toimivat yhtä tehokkaasti ja vertailuarvot eroavat vain keskiarvoltaan. Keskiarvon osoittama eroa ei kuitenkaan voi vähätellä; A*-haun sulkiessa ylimääräisiä solmuja noin 12 % parhaan ratkaisupolun pituuden mukaisesta määrästä, oppiva haku sulki ylimääräisiä solmuja noin 30 % samasta vertailuarvosta.

Hakutehtävän lähtö- ja kohdesolmujen välinen matka vaikuttaa A*-haun tehokkuuteen. Tämä ilmenee suljettujen solmujen määrän vertailuarvon kasvuna suhteessa hakupolun pituuteen, joka on esitetty kuvassa 22. Kuva osoittaa, että vain hyvin lyhyitä etäisyyksiä sisältävät hakutehtävät

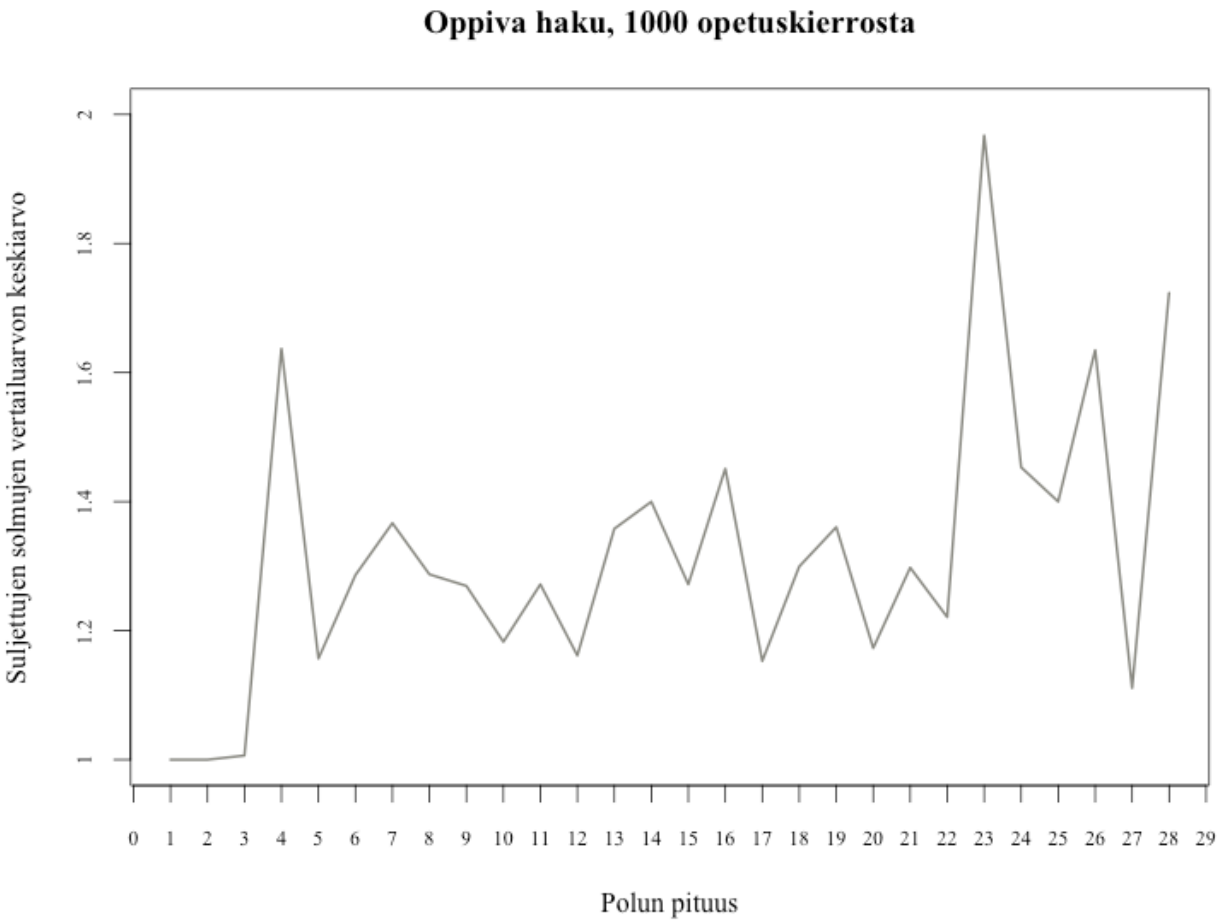
löydetään ilman ylimääräisiä suljettuja solmuja, mutta hakupolun kasvaessa, myös suljettujen solmujen suhteellinen osuus kasvaa. Tämän tiedon perusteella voidaan olettaa, että hakutehtävän lähtö- ja kohdesolmujen välinen etäisyys vaikuttaa myös oppivan haun tehokkuuteen.



Kuva 22. Polun pituuden suhde suljettujen solmujen määrään A*-haussa.

A*-haun suljettujen solmujen määrä kulkee optimaalisen suhdeluvun mukaan lyhyitä polkuja omaavissa hakutehtävissä, mutta polun pituuden kasvaessa suljettujen solmujen määrä kasvaa polun pituutta nopeammin. Polun pituuden ja suljettujen solmujen lukumäärän vertailuarvon välille saadaan Pearsonin korrelaatiokertoimeksi $r = 0{,}1419$ ($p < 0{,}001$). Tulos osoittaa, että polun pituuden ja suljettujen solmujen vertailuarvon välillä on positiivinen korrelaatio, vaikka korrelaatio onkin heikko. Tulos on tyydyttävä, sillä polun pituuden itsessään ei pitäisi vaikuttaa suljettujen solmujen suhteellisen määrän kasvuun, vaan todennäköisyyteen sille, että polun varrella on umpikujia, joista pois pääsemiseksi haun tulee tutkia ylimääräisiä solmuja.

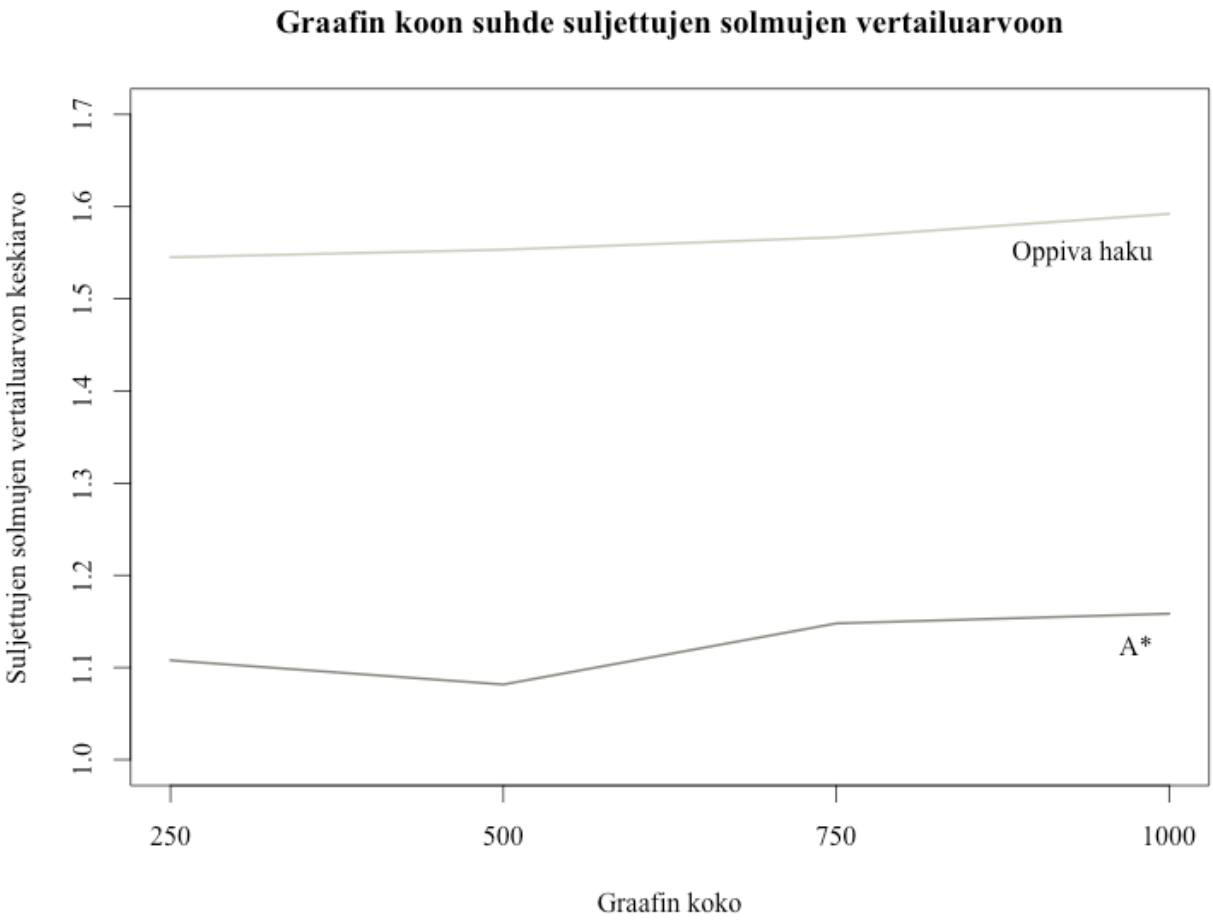
Kuva 23 esittää vastaavan kuvaajan suljettujen solmujen määrän ja polun pituuden suhteen oppivalle haulle 1000. opetuskierroksen jälkeen. Kuvan mukaan suljettujen solmujen määrän suhteellinen osuus on A*-haun vastaavia arvoja suurempi erityisesti lyhyissä hakutehtävissä, mutta toisin kuin A*-haun yhteydessä, suljettujen solmujen suhteellinen osuus ei näyttäisi kasvavan polun pituuden kasvaessa yhtä voimakkaasti kuin A*-haun yhteydessä.



Kuva 23. Oppivan haun ratkaisupolun pituuden suhde suljettujen solmujen määrän keskiarvoon.

Oppivan haun 1000. opetuskierroksen jälkeen laskettu Pearsonin korrelaatiokerroin on r = 0,0158 (p = 0,3171). Lähes olematon korrelaatiokerroin voidaan siis hylätä suuren p-arvon perusteella.

Samoin kuin polun pituuden, myös graafin solmujen määrän voidaan olettaa vaikuttavan hakualgoritmien tehokkuuteen. Hakualgoritmin toimiessa graafissa, jossa solmuja on vähän, on algoritmin mahdollista tutkia jokainen solmu järkevässä ajassa. Tästä syystä heikkoja hakualgoritmeja voidaan käyttää pienissä graafeissa. Heuristiset- ja oppivat hakualgoritmit tulevat heikkoja algoritmeja tehokkaammiksi graafin koon kasvaessa, sillä heikot hakualgoritmit tutkivat huomattavasti enemmän solmuja kuin ratkaisun löytäminen vaatii. Koska heikkojen hakualgoritmien tehokkuus kärsii siinä määrin, että ne tulevat lähes käyttökelvottomiksi suurissa graafeissa, voidaan myös heuristisen- ja oppivien hakujen olettaa kärsivän graafin koon kasvaessa.



Kuva 24. Suljettujen solmujen vertailuarvo kasvaa hitaasti graafin koon kasvaessa A*- ja oppivan haun yhteydessä.

Kuva 24 esittää suljettujen solmujen vertailuarvon keskiarvon muutoksen graafin koon kasvaessa. Vaikka suljettujen solmujen määrä näyttäisi kasvavan, on kasvu hyvin hidasta sekä A*- että oppivan haun yhteydessä. Kun graafin solmujen kokonaismäärä kasvaa nelinkertaiseksi, kasvaa suljettujen solmujen vertailuarvon keskiarvo noin 0,05 yksikköä molempien algoritmien yhteydessä.

Graafin koon ja A*-haun suljettujen solmujen vertailuarvon välinen Pearsonin korrelaatiokerroin on r = 0,0484 (p = 0,0025). Testistä saadun p-arvon perusteella korrelaatiokerroin on erisuuri kuin 0, mutta pienen korrelaatiokertoimen perusteella voidaan todeta, etteivät suljettujen solmujen vertailuarvot ja graafin koko varsinaisesti korreloi keskenään. Samaan johtopäätökseen tullaan oppivan haun yhteydessä, jossa Pearsonin korrelaatiokerroin on r = -0,0314 (p = 0,0470). Kuvan 24 esittämä suljettujen solmujen vertailuarvon kasvu vaikuttaakin olevan pikemmin yhteydessä siihen, että graafin koon kasvaessa myös parhaan ratkaisun keskipituus kasvaa. Tämä puolestaan vaikuttaa suljettujen solmujen määrään erityisesti A*-haun yhteydessä, kuten aiemmin todettiin.

7.2. Oppivan haun mittaaminen sosiaalista verkkoa kuvaavassa graafissa

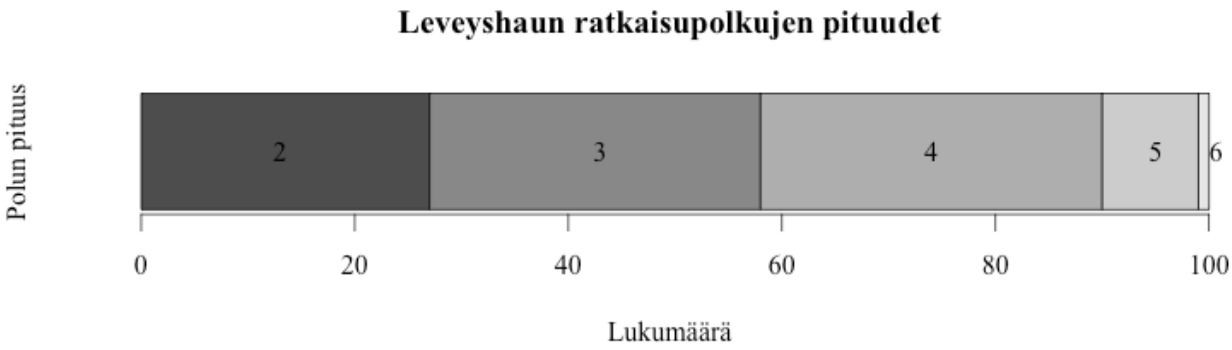
Sosiaalisten suhteita ja -vuorovaikutusta on usein luonnollista kuvata graafina, missä graafin solmut kuvaavat sosiaalisen verkon jäseniä ja kaaret jäsenien välisiä suhteita. Tästä syystä sosiaaliset verkot ovat luonnollinen käyttökohde graafialgoritmeille. Sosiaaliset verkot eivät kuitenkaan tarjoa luonnollista heuristiikkaa, jonka mukaan graafin solmujen välisiä polkuja voisi hakea heuristisen haun avulla, minkä vuoksi solmujen välisiä yhteyksiä haetaan heikoilla hakualgoritmeilla tai niistä johdetuilla algoritmeilla.

Wilson tutkimusryhmineen [2009] kokosi vuorovaikutusgraafeja 22 Facebookin suurimmasta alueellisesta sosiaalisesta verkosta (*regional network*). Facebookin alueelliset verkostot ovat yhteisöjä, joihin käyttäjät liitetään automaattisesti maantieteellisen sijainnin mukaan. Käyttäjäprofiilien hakeminen näistä verkostoista tuotti mahdollisimman heterogeenisen otoksen, jonka pohjalta tutkijat alkoivat koota sosiaalista vuorovaikutusta kuvaavia graafeja Facebook-profiileja tutkivan "ryömijän" avulla. Tutkimus esittää 10 suurimman verkon keskeiset arvot ryömijän kokoamien tietojen perusteella. Tutkimuksen mukaan 10 suurimpaan graafiin oli koottu yhteensä noin 10 697 tuhatta käyttäjää, jotka muodostivat 56,3% alueiden kaikista käyttäjistä. Kaaria graafeihin kuului yhteensä noin 408 265 tuhatta, mistä muodostui 43,3% alueiden kaikista vuorovaikutuksista. Solmujen ja kaarien määrien suhteesta voidaan päätellä, että graafien polkujen tutkiminen leveyshaun mukaan on epäkäytännöllistä.

Sosiaaliset verkot tarjoavat oppivalle haulle sopivan sovelluskohteen, sillä oppivan haun pitäisi oppia heikkoja hakualgoritmeja tehokkaampi toimintamalli. Sosiaalisiin verkkoihin ei myöskään ole olemassa sopivaa heuristista soveltuvuusfunktiota, joka pystyisi ohjaamaan hakualgoritmia kohti kohdesolmua, joten heuristiset haut eivät ole käyttökelpoisia sosiaalisten verkkojen yhteydessä, toisin kuin euklidiseen avaruuteen kuvatuissa graafeissa.

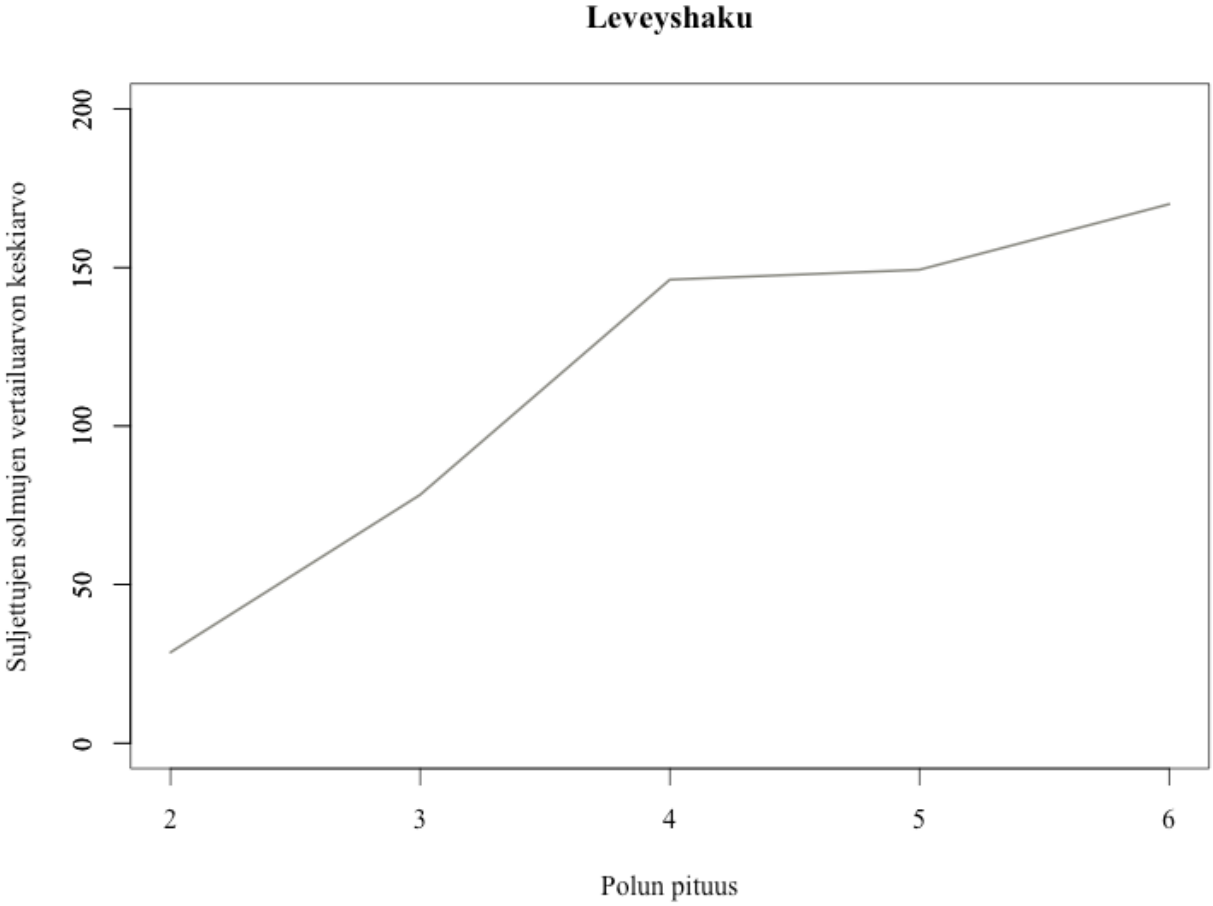
Oppivan haun toimintaa sosiaalisessa verkossa mitataan tässä tutkielmassa Leskovecin ja Krevlin ylläpitämässä Facebook-verkkoaineistossa [Leskovec & Krevl, 2014; McAuley & Leskovec, 2012]. Koska alkuperäinen aineisto koostuu useasta graafista, on tähän kokeeseen valittu yksi graafi, joka koostuu 1 034 solmusta ja 53 498 kaaresta. Koe koostuu sadasta hakutehtävästä, joiden lähtö- ja kohdesolmut on valittu satunnaisesti. Oppivan haun suoriutumista verrataan leveyshaun tuottamiin tuloksiin samoissa hakutehtävissä.

Kokeen aikana mitataan hakupolkujen pituutta ja suljettujen solmujen määrää samoin kuin ensimmäisessä kokeessa. Satunnaisesti generoituihin hakutehtäviin suoritettiin leveyshaku, joka tuottaa yhden parhaista mahdollisista ratkaisupoluista kuhunkin hakutehtävään. Vertailukelpoisena mittarina käytetään mitattujen arvojen suhdetta parhaan mahdollisen hakupolun pituuteen, joka saadaan leveyhaun tuottamasta ratkaisupolusta. Ratkaisupolkujen pituudet vaihtelivat kahden ja kuuden solmun välillä siten, että kahden solmun pituisia ratkaisupolkuja hakutehtävissä oli 27, kolmen solmun pituisia 31, neljän solmun pituisia 32, viiden solmun pituisia 9 ja kuuden solmun pituisia yksi. Ratkaisusolmujen pituuksien suhteellinen osuus kaikista hakutehtävistä on esitetty kuvassa 25.



Kuva 25. Hakutehtävien parhaan mahdollisen ratkaisupolun pituudet ja niiden määrä leveyshaun mukaan.

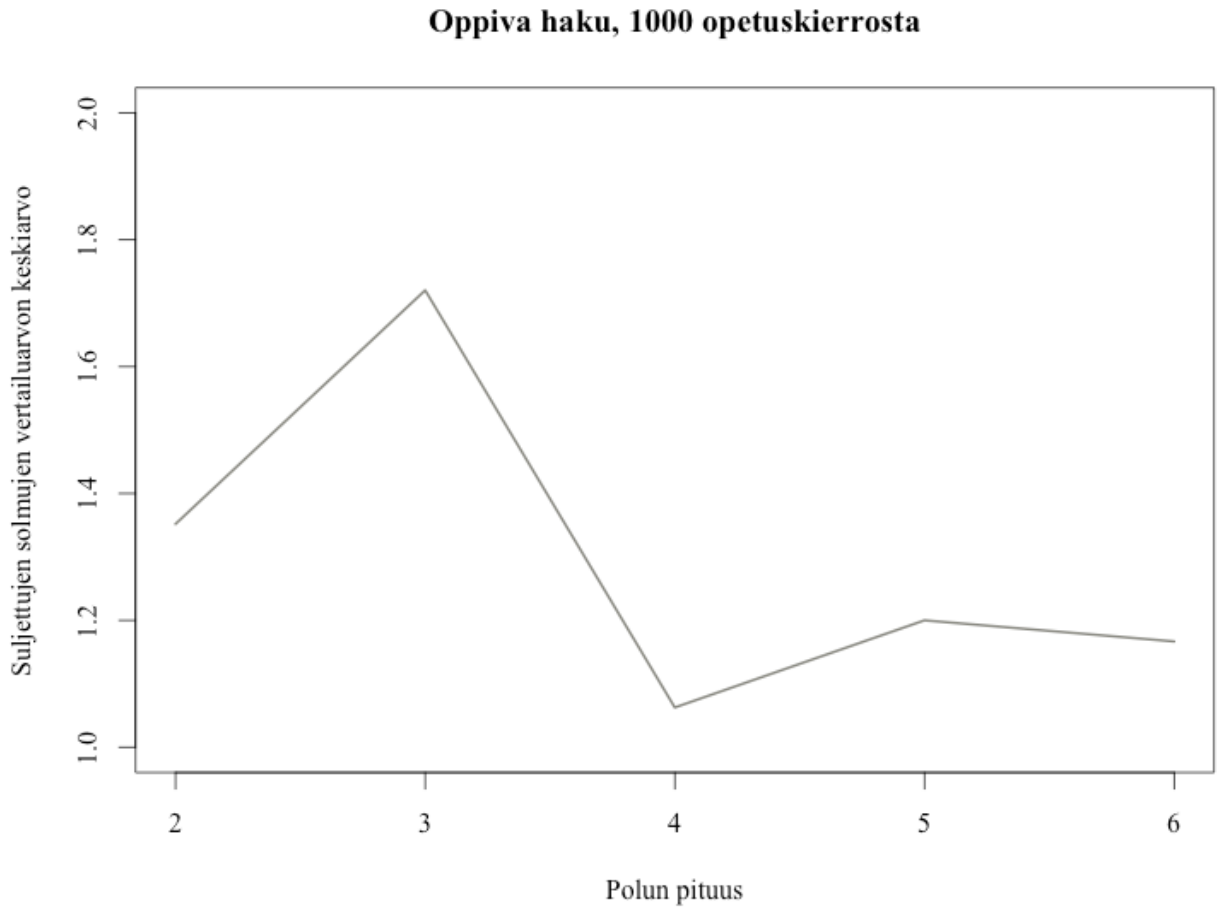
Koska leveyshaun tuottaman ratkaisupolun pituus on aina mahdollisimman lyhyt, on leveyshaun ratkaisupolun pituuden vertailuarvo aina yksi. Tämä noudattaa siis samaa ajatusta kuin A*-haun ratkaisupolun pituuden vertailuarvo ensimmäisessä kokeessa. Leveyshaun toimintaa voidaan tutkia tarkastelemalla leveyshaun sulkemien solmujen määrää. Kuvassa 26 on esitetty leveyshaun sulkemien solmujen vertailuarvot ratkaisupolun pituuden mukaan jaoteltuna.



Kuva 26. Leveyshaun sulkemien solmujen lukumäärän vertailuarvot polun pituuden mukaan.

Kuvasta 26 havaitaan leveyshaun taipumus tutkia suuria määriä solmuja matalissakin hakupuissa. Kun ratkaisupolku oli kahden solmun pituinen, leveyshaku sulki keskimäärin noin 28 kertaa enemmän solmuja ratkaisupolun pituuteen nähden. Ratkaisupolun pituuden ollessa neljä, leveyshaku sulki keskimäärin noin 146 kertaa enemmän solmuja ja kuuden solmun pituisessa ratkaisupoluissa solmuja suljettiin 170 kertaa ratkaisupolun kokoa enemmän. Kaikkien hakutehtävien vertailuarvon keskiarvo leveyshaun sulkemien solmujen määrälle on 129.

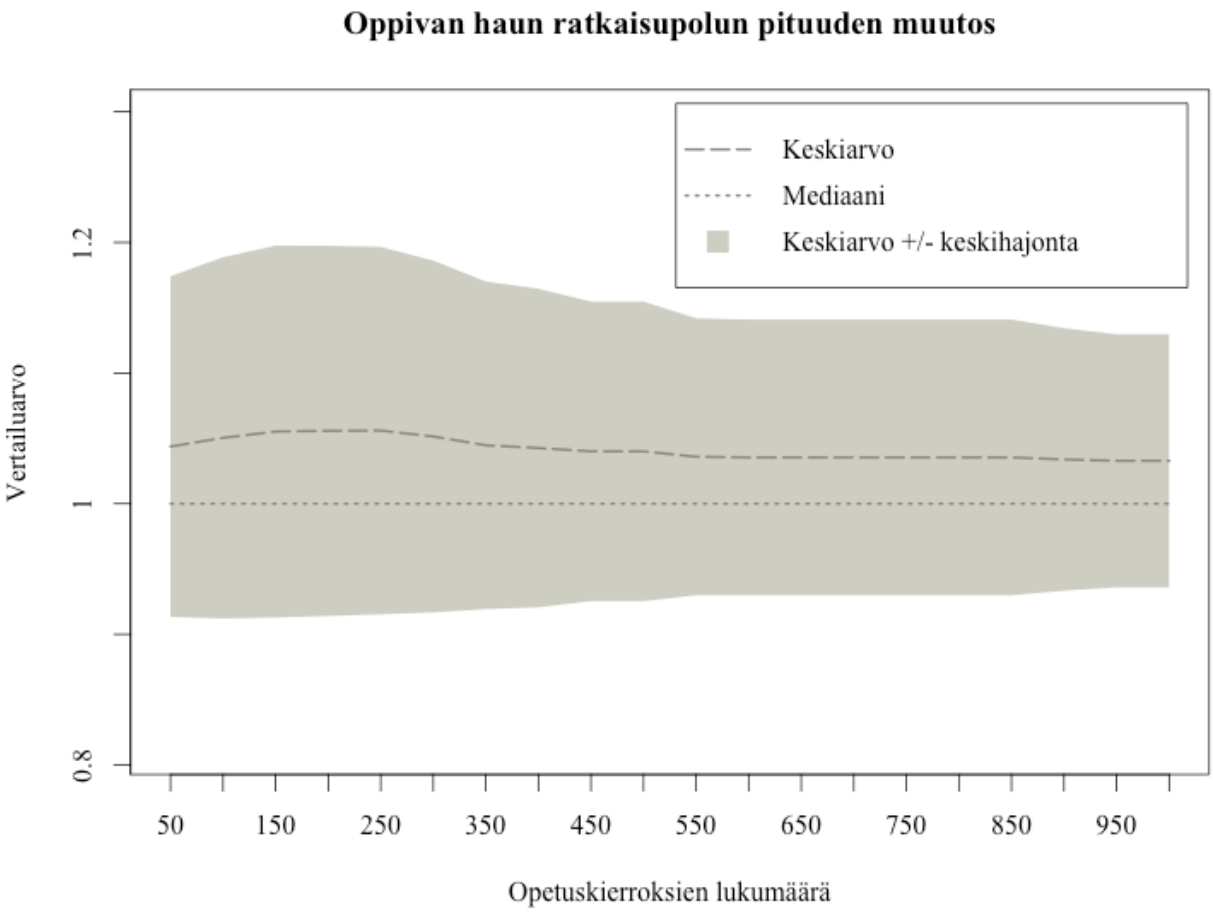
Oppivan haun sulkemien solmujen määrän vertailuarvo 1000 oppimiskierroksen jälkeen ratkaisupolun pituuden mukaan jaoteltuna on esitetty kuvassa 27. Kuva 27 osoittaa, että opittuaan hakutehtävään liittyvän aligraafin, oppiva haku pystyy löytämään ratkaisupolun erittäin tehokkaasti. Suljettujen solmujen määrän vertailuarvo on alle 2 kaikissa hakutehtävissä, mikä tarkoittaa, että oppiva haku sulkee korkeintaan kaksi kertaa enemmän solmuja kuin mitä ratkaisupolkuun tarvitaan. Kuvasta voidaan myös päätellä, että ratkaisupolun pituudella ei ole yhtä suurta vaikutusta oppivan haun suljettujen solmujen määrän vertailuarvoon kuin leveyshaun vastaavaan arvoon. Kaikkien hakutehtävien vertailuarvo oppivan haun sulkemien solmujen määrälle on 1,6855, mikä on huomattavasti vähemmän kuin leveyshaun vastaava arvo.



Kuva 27. Oppivan haun sulkemien solmujen määrän vertailuarvo ratkaisupolun pituuden mukaan jaoteltuna.

Oppivan haun sulkemien solmujen määrässä näkyy piikki kahden- ja kolmen solmun pituisten ratkaisupolkujen yhteydessä. Tämä johtuu todennäköisesti siitä, että ensimmäiset hakutehtävät sisälsivät kahden ja kolmen solmun pituisia ratkaisupolkuja. Tällöin kyseiset hakutehtävät suorittivat enemmän kokeilevia valintoja, koska graafista ei ollut valmiina käytettävissä olevaa tietoa. Kun hakutehtävistä siirryttiin seuraavaan, edellisen tehtävän tuottama tieto graafista säilyi agentin muodostamassa mallissa, minkä vuoksi seuraavat haut pystyivät suoriutumaan edellistä tehokkaammin.

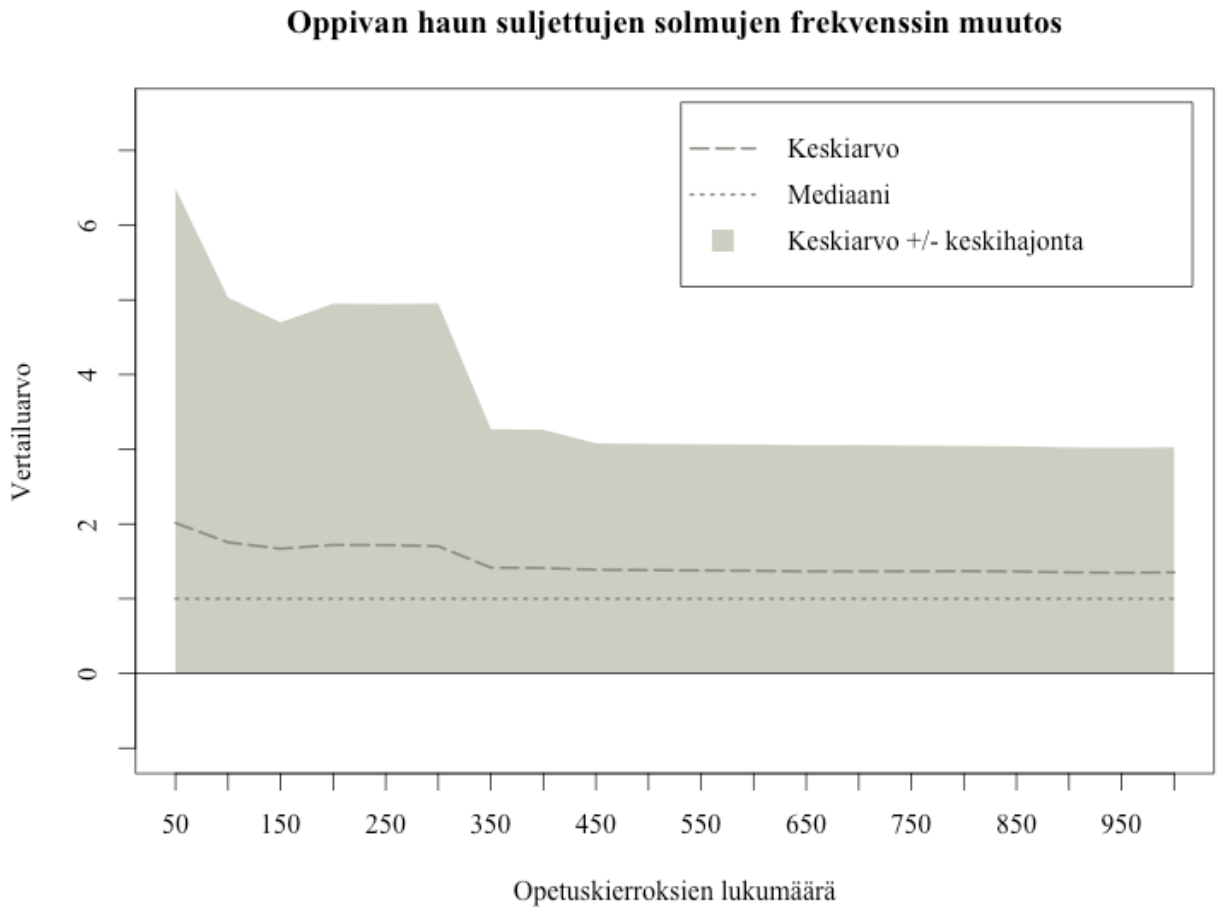
Oppivan haun tuottaman ratkaisupolun pituuden muutos oppimisprosessin aikana on esitetty kuvassa 28. Ratkaisun pituuden vertailuarvossa ei tapahdu suurta muutosta, sillä vertailuarvo on lähtökohtaisesti hyvällä tasolla. Pituuden vertailuarvon mediaani on 1,0000 läpi koko oppimis-vaiheen. Vertailuarvon keskiarvo on korkeimmillaan 1,0559 ja laskee 500 oppimiskierroksen jälkeen arvon 1,0400 alapuolelle. Huomattavin muutos nähdään pituuden vertailuarvon keskihajonnassa, joka on korkeimmillaan 0,1417 ja laskee 500 opetuskierroksen jälkeen arvoon 0,1056.



Kuva 28. Oppivan haun tuottaman ratkaisupolun pituuden vertailuarvon muutos oppimisen edetessä.

Ratkaisupolun pituuden muutos osoittaa, että haku kehittyy oppimisen myötä, mutta kehittyminen on melko vaatimatonta tämän mittarin mukaan. Sen sijaan haun tehokkuuden kehitys

tulee paremmin ilmi algoritmin sulkemien solmujen määrän vertailuarvon muutosta tutkimalla. Kuva 29 esittää oppivan haun sulkemien solmujen vertailuarvon kehityksen oppimisprosessin aikana.



Kuva 29. Oppivan haun sulkemien solmujen määrän vertailuarvon muutos oppimisprosessin aikana.

Samoin kuin ratkaisupolun pituuden määrän vertailuarvon kohdalla, myös vertailuarvon mediaani on 1,0000 läpi koko opetusvaiheen. Vertailuarvon keskiarvo muuttuu arvosta 2,0138 arvoon 1,3873 400 opetuskierroksen jälkeen. Vertailuarvon keskihajonta alkaa arvosta 4,4740 ja supistuu arvoon 1,8502 350 opetuskierroksen jälkeen.

Mittaustulosten perusteella voidaan oppivan haun todeta onnistuvan oppimistehtävässä, kun haku saavuttaa oppimisvaiheen päätteeksi lähtötilannetta tehokkaamman toimintamallin. Kokeessa käytetyssä graafissa pääosa oppimisen hyödyistä saavutettiin ensimmäisten 350 oppimiskierroksen aikana, jonka jälkeen oppimisvauhti hidastui huomattavasti. Leveyshakuun verrattuna oppiva haku saavutti erinomaisen oppimistuloksen; oppivan haun tuottaman ratkaisupolun pituus on valtaosassa hakutehtäviä paras mahdollinen, ja ratkaisupolun pituuksien vertailuarvon vaihteluväli on suppea. Oppivan haun sulkemien solmujen määrä on huomattavasti leveyshaun sulkemien solmujen määrää pienempi, minkä ansiosta oppivaa hakua voidaan pitää moninkertaisesti leveyshakua tehokkaampana.

8. Pohdintaa

Graafihakualgoritmit ovat yleisiä työkaluja graafirakenteisen tiedon tutkimisessa. Datamassojen jatkuva kasvu asettaa graafihakualgoritmeille jatkuvasti korkeampia tehokkuusvaatimuksia. Heikot graafihakualgoritmit tarjoavat toimintavarman ratkaisun graafihakuongelmaan, mutta käsittelevät usein suuren määrän ylimääräisiä solmuja. Heikkoja hakualgoritmeja on pyritty tehostamaan muokkaamalla itse algoritmia, kuten kahden rintaman leveyshaussa, tai haun kohteena olevaa graafia, kuten graafin ositusalgoritmeissa. Heikkoja hakuja muokkaamalla voidaan kuitenkin korkeintaan lykätä heikoille hakualgoritmeille ominaista ongelmaa – avointen solmujen määrän vauhdikasta kasvua.

Heuristiset haut tuovat merkittävän parannuksen graafihakuihin yksinkertaisella menetelmällä: avointen solmujen listan järjestämisellä. Järjestämällä avointen solmujen lista hakutehtävän kannalta suotuisasti voidaan hakutehtävästä suoriutua parhaassa tapauksessa käsittelemättä yhtäkään ylimääräistä solmua. Heuristisen hakujen heikkous on kuitenkin juuri avointen solmujen listan järjestävä soveltuvuusfunktio, sillä kaikkiin hakutehtäviin ei tällaista funktiota ole määriteltävissä. Tämä rajaa heuristisen hakujen sovelluskenttää huomattavasti. Jos hakutehtävään ei voida laatia heuristista hakumenetelmää, tyydytään tällöin usein heikkoihin hakumenetelmiin.

Koneoppimisen soveltaminen graafihaussa perustuu samaan motivaatioon kuin koneoppimisen soveltaminen missä tahansa muussa sovellusalueessa. Monimutkaisten mallien rakentaminen ja optimoiminen annetaan koneen oppimistehtäväksi, jolloin algoritmin toiminta tehostuu haastavissakin sovelluskohteissa. Koneoppimiseen perustuva graafihakumenetelmä jatkaa heuristisen hakujen kehityssuuntaa siten, että heuristisen soveltuvuusfunktion laatiminen annetaan koneelle oppimistehtäväksi. Tällöin ei pelkästään laajenneta heuristisen hakujen rajattua sovelluskenttää, vaan sen lisäksi luovutaan heuristisen soveltuvuusfunktion sovelluskohde-kohtaisesta räätälöintitarpeesta. Räätälöidyt heuristiset haut ovat usein käytettävissä vain yhdessä sovelluskohteessa sen erikoistuneen soveltuvuusfunktion vuoksi, mutta koneoppiva haku voi käyttää samaa oppimismenetelmää oppiakseen minkä tahansa sovelluskohteen soveltuvuusfunktion.

Tässä tutkielmassa on esitetty oppiva graafihakualgoritmi, joka pyrkii laajentamaan heuristisen hakualgoritmien sovelluskenttää soveltamalla vahvistusoppimista oppivan hakualgoritmin toteuttamisessa. Vahvistusoppimismenetelmä tarjoaa hyvän perustan oppivalle haulle, sillä menetelmä vaatii hyvin vähän tietoa sovelluskohteesta, kasvattaa opittua mallia samalla kun uusia solmuja havaitaan hakualgoritmin edetessä ja pystyy muokkaamaan jo opittua tietoa jos sovellusalueessa tapahtuu muutoksia.

Vahvistusoppimisen soveltaminen graafihakuun on erityisen kiinnostavaa siksi, että menetelmän avulla voidaan luopua heuristisen hakumenetelmien ja joidenkin nykyisten koneoppivien graafihakumenetelmien asettamasta vaatimuksesta solmujen piirteitä kohtaan. Siinä missä heuristiset graafihakumenetelmät vaativat solmuihin sisältyvän piirrevektorin, jonka perusteella soveltuvuusfunktio arvottaa solmut, vahvistusoppimiseen perustuva graafihaku-menetelmä vaatii vain solmun yksilöivän tunnisteen.

Oppivaa hakua on testattu kahdessa kokeessa: ensimmäisessä oppivaa hakua verrattiin erittäin hyviin tuloksiin pystyvään heuristiseen A*-hakumenetelmään satunnaisesti generoiduissa

graafeissa. Toisessa kokeessa oppivan haun suoriutumista tutkittiin sosiaalista verkkoa kuvaavassa graafissa, johon ei ole hyvin määritettyä heuristista soveltuvuusfunktiota.

Ensimmäisen kokeen mittaustulosten mukaan oppiva haku ei täysin saavuta A*-hakua vastaavaa tehokkuustasoa annetun 1000 opetuskierroksen aikana. A*-haku on siis kokeiden perusteella oppivaa hakua tehokkaampi. Oppivan haun ratkaisupolun pituuden ja suljettujen solmujen määrän vertailuarvojen mediaanit osoittavat kuitenkin suuren osan oppivan haun ratkaisuista saavuttavan parhaan mahdollisen tason, joten oppivaa hakua voidaan pitää kilpailukykyisenä hakumenetelmänä heuristisiin hakuihin nähden. Lisäksi oppivan haun tehokkuus paranee oppimisajojen myötä, ja lopullista tasoa ei 1000 opetusajon aikana saavutettu, vaikka oppimisnopeus hidastui huomattavasti.

Vaikka oppiva haku sulki keskimäärin enemmän solmuja kuin A*-haku, polun pituuden ja suljettujen solmujen määrän välillä ei oppivan haun yhteydessä havaittu korrelaatiota. Heuristisella haulla havaittiin korrelaatio kyseisten muuttujien välillä, minkä perusteella voidaan olettaa, että polun pituuden kasvaessa riittävän suureksi, tulee oppiva haku sulkemaan vähemmän solmuja kuin heuristinen haku samassa hakutehtävässä. Tämä olettamus jää kuitenkin tämän tutkielman yhteydessä vahvistamatta ja jäänee tulevien tutkimusten käsiteltäväksi.

Toisen kokeen mittaustulosten mukaan oppiva haku saavuttaa suuressa osassa hakutehtäviä parhaan mahdollisen ratkaisupolun ja suljettujen solmujen määrä on melko pieni. Oppivaa hakua verrattiin leveyshakuun, joka on usein käytetty hakumenetelmä sosiaalisten verkkojen yhteydessä. Sosiaalisten verkkojen yhteydessä leveyshaku on varma tapa tuottaa ratkaisupolku hakutehtävään, mutta menetelmään liittyy paljon tehottomuutta. Kokeen hakutehtävissä leveyshaku sulki keskimäärin 129 kertaisen määrän solmuja suhteessa parhaaseen mahdolliseen tulokseen. Oppivan haun vastaava tulos oli alle 2. Tämä tulos osoittaa oppivan haun pystyvän tuomaan heuristisiin hakuihin verrattavan tehokkuustason sovelluskohteisiin, joihin on vaikeaa tai mahdotonta laatia heuristista hakua perinteisin keinoin.

Kokeiden tuottamat mittaustulokset ovat lupaavia, mutta oppivalla haulla on myös heikkoutensa. Tässä tutkielmassa esitetyn oppivan hakumenetelmän oppiva osuus perustuu naiivisti toteutettuun vahvistusoppimismenetelmään, joka tallentaa jokaisen malliin kuuluvan arvon erikseen avain–arvoparina. Vahvistusoppimisen tutkimuksessa tämä lähestymistapa on todettu ongelmalliseksi suuren muistivaatimuksen vuoksi, sillä oppimistehtävän tulee oppia sopiva arvo jokaiselle agentin toiminnolle jokaisessa ympäristön tilassa. Tämä tarkoittaa karteesista tuloa $A \times S$ agentin toimintojen A ja ympäristön tilojen S välillä. Tässä tutkielmassa esitetty oppiva haku menee vielä pitemmälle, sillä haun tulee oppia sopiva arvo jokaiselle solmulle jokaisessa graafin solmussa jokaista kohdesolmua kohden. Tämä tarkoittaa karteesista tuloa $S \times A \times S$ graafin solmujen S, kunkin solmun naapurisolmun A ja jokaisen kohdesolmun S välillä. Esitetyn oppivan hakumenetelmän keskeinen ongelma kumpuaa siis vahvistusoppimisen alalta, missä kyseistä ongelmaa on pyritty hillitsemään soveltamalla erilaisia funktion approksimointimenetelmiä. Oppivan haun jatkokehityksessä voidaankin tutkia vahvistusoppimismenetelmän soveltuvuutta oppivan haun toteutuksessa ja mahdollisia vaihtoehtoja valitulle menetelmälle sekä funktion approksimointi-menetelmien, kuten neuroverkkojen, tehokkuutta oppivan haun kontekstissa.

Viiteluettelo

[Andrieu *et al.*, 2003] Christophe Andrieu, Nando de Freitas, Arnaud Doucet & Michael I. Jordan, An Introduction to MCMC for Machine Learning. *Machine Learning* **50** (2003), 5-43.

[Arfaee *et al.*, 2011] Shahab Jabbari Arfaee, Sandra Zilles & Robert C. Holte, Learning heuristic functions for large state spaces. *Artificial Intelligence* **175** (2011), 2075-2098.

[Auer *et al.*, 2002] Peter Auer, Nicoló Cesa-Bianchi & Paul Fischer, Finite-time analysis of the multiarmed bandit problem. *Machine Learning* **47** (2002), 235-256.

[Beamer *et al.*, 2013] Scott Beamer, Krste Asanović and David Patterson, Direction-optimizing breadth-first search. *Scientific Programming* **21** 3-4 (2013), 137-148.

[Chen *et al.,* 2014] Weihai Chen, Haosong Yue, Xingming Wu & Jianhua Wang, Real-time obstacle detection for legged robots using the Kinect sensor. *Advanced Robotics* **28**, 20 (2014), 1375–1387.

[Duguleana & Mogan, 2016] Mihai Duguleana & Gheorghe Mogan, Neural networks based reinforcement learning for mobile robots obstacle avoidance. *Expert Systems with Applications* **62** (2016), 104-115.

[Fathinezhad *et al.*, 2016] Fatemeh Fathinezhad, Vali Derhami & Mehdi Rezaeian, Supervised fuzzy reinforcement learning for robot navigation. *Applied Soft Computing* **40** (2016), 33-41.

[Fay, 2016] Damien Fay, Predictive partitioning for efficient BFS traversal in social networks. *Proceedings of the 7th Workshop on Complex Networks* **644** (2016), 11-26.

[Feng & Tan, 2016] Shu Feng & Ah-Hwee Tan, Towards autonomous behavior learning of non-player characters in games. *Expert Systems with Applications* **56** (2016), 89-99.

[Fink, 2007] Michael Fink, Online learning of search heuristics. *Proceedings of the 11th International Conference on Artificial Intelligence and Statistics (AISTATS-07)* **2** (2007), 114-122.

[Ferretti *et al*., 2016] Stefano Ferretti, Silvia Mirri, Catia Prandi & Paola Salomoni, Automatic web content personalization through reinforcement learning. *The Journal of Systems and Software* **121** (2016), 157-169.

[Guzolek & Koch, 1989] John Guzolek & Edward Koch, Real-time route planning in road networks. *Vehicle Navigation and Information Systems* (1989), 165-169.

[Hart *et al.*, 1968] Peter E. Hart, Nils J. Nilsson & Bertram Raphael, A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* **4**, 2 (1968), 100-107.

[Jaakkola *et al.*, 1994] T. Jaakkola, M. I. Jordan, & S. P. Singh, On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation* **6**, 6 (1994), 1185-1201.

[Kocsis & Szepesvári] Levente Kocsis & Csaba Szepesvári, Bandit based Monte-Carlo planning. *Proceedings of the 17th European Conference on Machine Learning* **4212** (2006), 282-293.

[Kreher & Stinson, 1999] Donald L. Kreher & Douglas R. Stinson, *Combinatorial Algorithms: Generation, Enumeration and Search.* CRC Press LLC, 1999.

[Lee *et al.*, 2016] Eun Kyung Lee, Hariharasudhan Viswanathan & Dario Pompili, RescueNet: reinforcement-learning-based communication framework for emergency networking. *Computer Networks* **98** (2016) 14-28.

[Leskovec & Krevl, 2014] Jure Leskovec & Andrej Krevl, SNAP Datasets: Stanford Large Network Dataset Collection. N.p., 2014. (http://snap.stanford.edu/data).

[Ling *et al.*, 2015] Mee Hong Linga, Kok-Lim Alvin Yaua, Junaid Qadirb, Geong Sen Pohc & Qiang Nid, Application of reinforcement learning for security enhancement in cognitive radio networks. *Applied Soft Computing* **37** (2015), 809-829.

[McAuley & Leskovec, 2012] Julian J. Mcauley & Jure Leskovec, Learning to discover social circles in ego networks. *Advances in Neural Information Processing Systems* **25** (2012), 539-547.

[Mocanu *et al.*, 2016] Elena Mocanu, Phuong H. Nguyen, Wil L. Kling & Madeleine Gibescu, Unsupervised energy prediction in a Smart Grid context using reinforcement cross-building transfer learning. *Energy and Buildings* **116** (2016), 646-655.

[Nanopoulos & Manolopoulos, 2001] Alexandros Nanopoulos & Yannis Manolopoulos, Mining patterns from graph traversals. *Data & Knowledge Engineering* **37** (2001), 243-266.

[Pearl, 1984] Judea Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley Publishing Company, Inc., Reading, MA, 1984.

[Petrik & Zilberstein, 2008] Marek Petrik & Shlomo Zilberstein, Learning heuristic functions through approximate linear programming. *Proceedings of the 18th International Conference on International Conference on Automated Planning and Scheduling* (2008), 248-255.

[Robbins, 1952] Herbert Robbins, Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society* **58** (1952), 527-535.

[Silvela & Portillo, 2001] Jaime Silvela & Javier Portillo, Breadth-first search and its application to image processing problems. *IEEE Transactions on Image Processing* **10**, 8 (2001), 1194-1199.

[Spurlock & Souvenir, 2016] Scott Spurlock & Richard Souvenir, Dynamic view selection for multi-camera action recognition. *Machine Vision and Applications* **27**, 1 (2016), 53-63.

[Stamatelos *et al.*, 2014] Spyros K. Stamatelos, Eugene Kim, Arvind P. Pathak & Aleksander S. Popel, A bioimage informatics based reconstruction of breast tumor microvasculature with computational blood flow predictions. *Microvascular Research* **91** (2014), 8-21.

[Stepanov & Smith, 2012] Alexander Stepanov & James MacGregor Smith, Modeling wildfire propagation with Delaunay triangulation and shortest path algorithms. *European Journal of Operational Research* **218** (2012), 775-788.

[Sutton & Barto, 1998] Richard S. Sutton & Andrew G. Barto, *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998.

[Tamir & Kandel, 1995] Dan E. Tamir & Abe Kandel, Logic programming and the execution model of Prolog. *Information Sciences - Applications* **4**, 3 (1995), 167-191.

[Thornton & du Boulay, 1998] Cristopher Thornton & Benedict du Boulay, *Artificial Intelligence: Strategies, Applications and Models Through Search.* Glenlake Publishing Company, Ltd., Chicago, IL, 1998.

[Treleaven *et al.*, 2005] Philip C. Treleaven, Apostolos N. Refenes, Kenneth J. Lees & Stephen C. McCabe, Computer architectures for artificial intelligence. *Lecture Notes in Computer Science* **272** (2005), 416-492.

[Wang & Lee, 2011] Yao-Te Wang & Anthony J.T. Lee, Mining web navigation patterns with a path traversal graph. *Expert Systems with Applications* **38** (2011), 7112-7122.

[Walraven *et al.*, 2016] Erwin Walraven, Matthijs T.J. Spaan & Bram Bakker, Traffic flow optimization: A reinforcement learning approach. *Engineering Applications of Artificial Intelligence* **52** (2016), 203-212.

[Wilson *et al.*, 2009] Christo Wilson, Bryce Boe, Alessandra Sala, Krishna P. N. Puttaswamy & Ben Y. Zhao, User Interactions in Social Networks and their Implications. *Proceedings of the 4th ACM European Conference on Computer Systems* (2009), 205-218.

[Xia & Zhao, 2016] Zhongpu Xia & Dongbin Zhao, Online reinforcement learning control by Bayesian inference. *IET Control Theory & Applications* **10**, 12 (2016), 1331-1338.